Office of Graduate Studies

Dissertation / Thesis Approval Form

This form is for use by all doctoral and master's students with a dissertation/thesis requirement. Please print clearly as the library will bind a copy of this form with each copy of the dissertation/thesis. All doctoral dissertations must conform to university format requirements, which is the responsibility of the student and supervising professor. Students should obtain a copy of the Thesis Manual located on the library website.

Dissertation/Thesis Title: Efficient Scaling of Out-of-Order Processor Resources

Author: Steven James Battle

This dissertation/thesis is hereby accepted and approved.

Signatures:

Examining Committee

Chair _____

Members _____

_____

_____

_____

_____

Academic Advisor _____

Department Head _____

Office of Graduate Studies • 3141 Chestnut St. • Randell Hall 240 • Philadelphia, PA 19104
Tel.: 215-895-0366 • **Fax**: 215-895-0495 • **Email**: graduate@drexel.edu • **Web**: www.drexel.edu/provost/graduatestudies

Efficient Scaling of Out-of-Order Processor Resources

A Thesis

Submitted to the Faculty

of

Drexel University

by

Steven James Battle

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy

June 2015

Dedications

This thesis is dedicated to my wife, Erica, my constant in a world full of variables.

Acknowledgments

First, I would like to thank my advisor Mark Hempstead for all of his support and encouragement as my advisor. His vision and enthusiasm for his research at Drexel helped convince me to begin this journey and return to academia five years ago. He took a chance on me and I am forever indebted to him for believing in me and guiding me on this journey to a PhD. His sage advice, optimism, experience, and criticisms helped drive our projects to completion and turn good papers into great papers.

Dr. Drew Hilton has been an almost constant presence throughout my research projects at Drexel (and afterwards at IBM). His guidance, expertise, and patience was critical in helping me complete this dissertation. His insightful, probing questions and fruitful discussions were essential to realizing the SCREW microarchitecture. I am grateful for his time and assistance.

I am indebted to my colleagues at Drexel who have been a tremendous resource for ideas, advice, and friendship. My cohort Siddarth Nilakantan has been a great sounding board for ideas and advice. I always enjoyed collaborating with Sid, as I could count on both his tireless work ethic and good humour as we toiled away towards late night conference deadlines. The rest of the Power Aware Computing lab: Rizwana Begum, Paco Sangaiah, Jason Palaszewski, Raymond Zhang, and Jonathan Stokes were great to work with and provided invaluable support and perspective as we worked to finish research projects and write papers. I also wish to thank my proposal and dissertation committees. Dr. Baris Taskin, Dr. Timothy Kurzweg, Dr. Naga Kandasamy, and Dr. Jeremy Johnson, Dr. Kapil Dandekar, provided insightful advice and helped shape my research and this dissertation.

I would not have enjoyed my return to academia nearly as much as I did without the support of my friends and family. I am lucky to have friends who looked out for me and endured my transition back to being a student. My friends were critical to both my physical and mental well-being, and the marathons we ran together helped prepare me for this long journey to a PhD. Training and

run-clubs helped to exercise my body and exorcise problems from my mind. My friends celebrated my achievements with me and picked me up whenever I needed help, and I am forever grateful.

My family has been my tireless cheerleaders and I am thrilled that they have always supported me. From a young age, my parents, Nella and Dave, encouraged me to continue learning and always took an interest in my work. My brothers, Andrew, Robert, and Tom were an emotional lifeline, and always able to pick me up when I needed it. I'd also like to thank my in-laws, Celia and Gary, along with Brian, for all of their support, advice, and encouragement during this journey. Finally, none of this would be possible without my wife and best friend, Erica. Her love, support, humour, and friendship mean the world to me and I look forward to the next steps on our journey together.

Table of Contents

List of Tables

List of Figures

Abstract

Efficient Scaling of Out-of-Order Processor Resources
Steven James Battle
Mark Hempstead, Ph.D.

Rather than improving single-threaded performance, with the dawn of the multi-core era, processor microarchitects have exploited Moore's law transistor scaling by increasing core density on a chip and increasing the number of thread contexts within a core. However, single-thread performance and efficiency is still very relevant in the power-constrained multi-core era, as increasing core counts do not yield corresponding performance improvements under real thermal and thread-level constraints.

This dissertation provides a detailed study of register reference count structures and its application to both conventional and non-conventional, latency tolerant, out-of-order processors. Prior work has incorporated reference counting, but without a detailed implementation or energy model. This dissertation presents a working implementation of reference count structures and shows the overheads are low and can be recouped by the techniques enabled in high-performance out-of-order processors. A study of register allocation algorithms exploits register file occupancy to reduce power consumption by dynamically resizing the register file, which is especially important in the face of wider multi-threaded processors who require larger register files.

Latency tolerance has been introduced as a technique to improve single threaded performance by removing cache-miss dependent instructions from the execution pipeline until the miss returns. This dissertation introduces a microarchitecture with a predictive approach to identify long-latency loads, and reduce the energy cost and overhead of scaling the instruction window inherent in latency tolerant microarchitectures. The key features include a front-end predictive slice-out mechanism and in-order queue structure along with mechanisms to reduce the energy cost and register-file usage of executing instructions. Cycle-level simulation shows improved performance and reduced energy delay for memory-bound workloads. Both techniques scale processor resources, addressing register file inefficiency and the allocation of processor resources to instructions during low ILP regions.

Chapter 1: Introduction

Processor performance in the multi-core era has largely been defined by three phenomena: the ever-present gap between CPU and memory performance known as the "memory wall" [92], increased pressure on off-chip memory bandwidth known as the "bandwidth wall" [71], and the increasing power-density and cooling requirements due to the end of Dennard scaling, known as the "power wall" [22, 30].

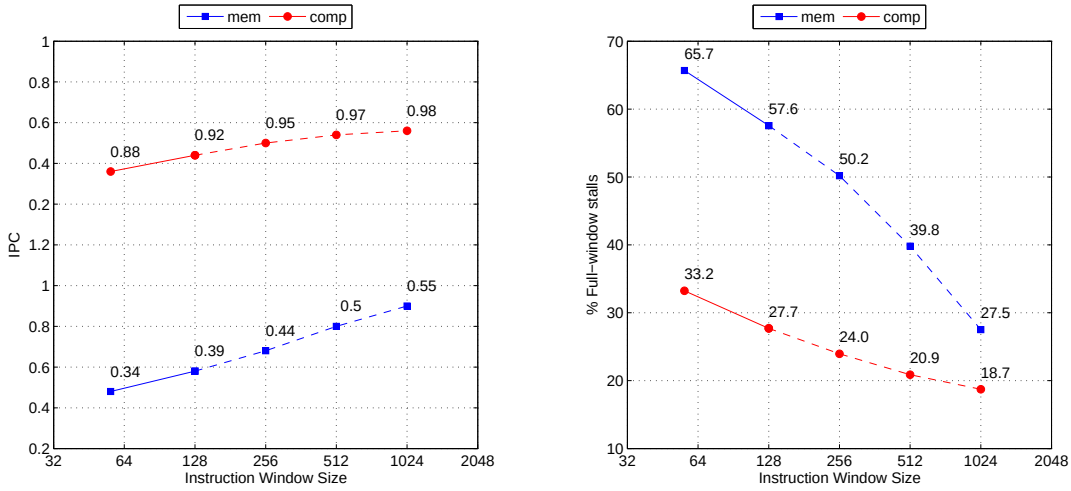
In this era, attention has shifted from improving the performance of a single execution thread performance through conventional device and design scaling, to focusing on chip multi-processors *(CMPs)* containing multiple cores with many execution threads on a single chip. It is assumed that software workloads will make use of the available parallelism in the system, in which increasing the CMP core count corresponds to increased performance. Typically, server and scientific workloads have abundant parallelism [1, 6] and metrics such as request bandwidth and throughput are more important than single-threaded performance [16, 39, 85]. However, increasing the core count in a CMP does not improve performance boundlessly, even for embarrassingly parallel workloads. Real-world systems are still physically constrained by power-density (the power-wall) and off-chip bandwidth (the bandwidth wall), both of which limit the amount of exploitable thread-level parallelism (TLP). In such cases, CMPs will again be limited by single-threaded performance [32, 77] and core energy-efficiency [21, 29, 31]. To this end, this dissertation focuses on techniques that improve performance and energy-efficiency of single-threaded cores. Improving single-threaded performance will reduce serial execution time *and* execution time of parallel sections of multi-threaded workloads. These improvements will be applicable to all general purpose processors, whether they are CMPs or stand-alone processors.

The traditional approach to improve single-threaded performance, making processor core structures bigger and clocking them faster [36], no longer applies in this era. Increasing the clock frequency only increases the relative performance gap between CPU core and memory, which means that when

a load cannot fetch its data from on-chip caches the processor will idle for many cycles while accessing main-memory [92], up to 195 cycles for Intel Nehalem processors [49]. Traditional voltage and frequency scaling (DVFS) is no longer the architect's panacea to improve performance *or* reduce power consumption. Voltage scaling to save energy is difficult, as chip voltage levels cannot be reduced without interfering with correct circuit operation [67], increasing sensitivity to process variation and temperature [7], or dramatic performance loss [18]. On the other hand, processors are already being shipped in which thermal limits prevent all cores from operating at maximum frequency at the same time [10, 22]. New cores must find novel ways to scale performance and energy. In this dissertation, we leverage the microarchitecture to reduce the energy cost of the register-file, applying microarchitecture information to circuit-techniques that reduce the static energy cost of large register-files and employ prediction mechanisms with a techniques for scaling the instruction window to large sizes.

1.1 Tolerating Latency

The classic out-of-order execution processor has been the quintessential design developed to tolerate long latency accesses to DRAM and reduce the effect of the "memory wall." An out-of-order processor executes instructions selected from a fixed-size "instruction window" [61]. Instructions enter the window in program-order at the front-end of the pipeline and are retired in program-order at the tail end of the pipeline. Instructions from within this window are able to be scheduled and executed out-of-order as soon as their operands are ready. A processor needs a large instruction window in order to find independent instructions to extract both instruction level parallelism (ILP) and memory level parallelism (MLP) from an application and tolerate long-latency cache misses. Instruction level parallelism, *ILP*, is exploited when instructions with no data dependencies between them execute in parallel. Memory level parallelism, *MLP*, occurs when multiple memory accesses are scheduled in parallel, where the latency of each access is overlapped rather than encountered sequentially. A small instruction window acts as a bottleneck during a cache-miss event, as it will be full of instructions which are miss-dependent, *i.e.* one operand depends on the load-miss or depends on an instruction that depends on the load-miss. A large instructions window allows the scheduler



(a) IPC

(b) % full instruction window stalls

Figure 1.1: Scaling the instruction window. Sweeping window size from 56- to 1024-instructions. All structures constraining the instruction window are scaled together (ROB, RF, IQ, LQ, SQ). SPEC benchmarks are classified as memory intensive (*mem*) or computation intensive (*comp*). The solid line indicates instruction window sizes achievable by conventional processor architectures, while dashed lines indicate large-window processors

to find more independent instructions from an application's instruction stream.

Figure 1.1 illustrates the effect of the instruction window size on workload performance. In Figure 1.1a, the instruction-window starts at 56 instructions with structures of the same size as in the small out-of-order AMD Bobcat processor [8]. We scale the instruction-window and each structure in the processor to support up to 1024 instructions in-flight, the size of 'kilo-instruction' processors [15, 35, 45]. The solid line identifies the instruction-window sizes of existing microarchitectures: including Bobcat, and Intel's Nehalem as current out-of-order processors have instruction-windows smaller than 256 instructions. The dashed line in the figure requires special techniques to achieve without exceeding thermal design and performance constraints. The workloads from SPEC2006 are split into two characteristic groups: computationally intensive and memory intensive. Computationally intensive workloads have higher ILP, fewer cache misses-per-thousand instructions *(MPKI)* than memory intensive workloads.

Figure 1.1a shows that for computationally intensive workloads, a 20× increase in window size

yields only an 11% improvement in performance. Memory intensive workloads show huge opportunity for improvement, with a 1024-entry window providing a 62% improvement on average, and greater than 100% for some benchmarks such as *lbm*, *milc*, and *soplex*. Figure 1.1b measures the fraction of dispatch stalls caused by the re-order buffer being fully occupied. As the instruction window structures grow, the fraction of stall drops for both types of workloads. Memory-bound instructions are particularly constrained by the re-order buffer as long-latency memory accesses stall the processor and cause the instruction window to fill. These workloads have the most to gain by scheduling instructions from a larger instruction window. A larger window allows the scheduler to find independent instructions to insert into the pipeline and make forward progress, while a smaller instruction window will be full of instructions waiting for the memory access to complete.

When data cannot be fetched from the last-level on-chip cache, a conventional in-order processor must idle, waiting for data to be read from memory before executing younger instructions. The defining feature of an out-of-order processor is that it does not need to pause execution when a cache-miss occurs. An out-of-order processor can continue to perform useful work and absorb the miss-latency by executing instructions from the instruction stream which do not depend on the load-miss. This instruction re-ordering allows the processor to overlap execution of many independent instructions simultaneously, extracting instruction level parallelism (ILP) from the workload. When multiple cache-misses are overlapped simultaneously, the processor is extracting memory level parallelism (MLP), significantly improving performance. However, out-of-order scheduling does not come for free; it requires several hardware structures to manage the instruction-window, track instruction dependencies and instruction ordering, and support precise-exceptions and branch speculation recovery [81]. These structures are critical to both the performance and energy consumption of the processor core and often do not scale well. More importantly, the in-order retirement constraint means that a long-latency instruction will block instructions from retiring, preventing younger instructions from entering the window, constraining performance.

There have been several approaches to scale the instruction window for out-of-order processors. These can be broadly split into two schema: designs that scale processor structures by making

them physically larger, and designs that scale structures by using them more efficiently. Scaling queues, buffers, and register-files to enormous sizes has been shown to increase processor performance [2, 43, 45, 70], leading to 'kilo-instruction' processors whose instruction window can be thousands of instructions wide. However, these studies often discount the area and power-cost of scaling these structures in hardware. To realistically achieve improved performance by physically scaling hardware, a designer needs very large associative-queues and register files that must either be accessed quickly or with less energy than conventional scaled designs [51, 64, 68], or radical new structures which function as both queue and register file [26]. These new structures often add complexity, latency, or consume large amounts of energy.

Latency-tolerant architectures such as CFP (Continual-Flow Pipelines) [82] and BOLT (Better Out-of-Order Latency Tolerance) [35] scale the instruction window by removing processor resources from instructions that cannot make forward progress. These architectures temporarily remove cache-miss dependent instructions from the instruction-window, releasing consumed resources (physical registers, issue-queue slots, ROB entries, etc.) until the cache-miss data is ready. However, these approaches also introduce their own overheads on resource management and instruction scheduling, and their structures may not always be energy-efficient. For example, latency tolerant architectures such as BOLT and CFP perform expensive value copy operations, shifting data out of the register file and into the slice-buffer and ROB. In some cases [45, 35], instructions can shuffle between queues before finally re-executing. In run-ahead [56], a scheme implemented in commercial processors instructions must be re-executed multiple times, even if they do not depend on the original load-miss! Existing techniques react to low-ILP events and scale resources by moving instructions from one buffer to another. In this dissertation, we present a technique to predict these low-ILP events and steer instructions out of the critical path *before* they become a bottleneck.

Out-of-order superscalar processors were historically found only in high-performance computing environments, but are now used in a diverse range of energy-constrained applications from smart-phones to data-centers. Thread-level parallelism alone is not the answer to scaling performance, as constraints in power, memory latency, and off-chip bandwidth are exacerbated by increasing the

number of cores on a chip. Improving single-threaded performance can provide relief and allow performance to scale, *if* it can be achieved in an energy-efficient manner. This dissertation addresses inefficiencies in scaling out-of-order processor resources in two ways: (i) by coupling microarchitecture and circuit techniques to relieve register-file energy pressure, and (ii) by scaling performance through a novel latency-tolerant microarchitecture.

1.2 Thesis Statement

Predicting low-ILP events and coupling low-power circuit techniques with micro-architectural information can effectively scale the out-of-order instruction window to tolerate long cache-miss latencies and improve single-threaded performance in an energy-efficient manner.

1.3 Dissertation Contributions

The work presented in this dissertation focuses in three areas: register management to enable efficient latency-tolerant designs, applying these register management microarchitecture techniques to reduce register-file power, and a novel energy-efficient approach to latency-tolerance. This dissertation contributes the following:

- **Detailed implementation and evaluation of register reference counting structures.** This dissertation shows that register reference counting bitvectors can be implemented with 7% increase in area and 0.8% increase in average power compared to conventional register management structures, and can support super-scalar architectures and designs supporting multiple reference-counting structures for SMT, checkpoint processing, and register sharing.

- **Reference-count inspired register file power-gating mechanism.** This dissertation shows that power-gating is a natural extension to register reference counting, presenting a circuit-level implementation of power-gating to connect the reference-count state information with a register-file gating controller.

- **Micro-architecture and circuit techniques for register-file power management** This dissertation presents several algorithms for allocating registers to the register-file in order to

maximize opportunity for power-gating. Also shown are algorithms and circuit techniques for improving register-file energy efficiency by taking advantage of "slack" inherent in the architecture. This dissertation shows how information from the microarchitecture can inform circuit-level techniques and improve energy efficiency.

- **Predictive Load Latency Tolerance** This dissertation presents a novel approach to latency tolerance - applying branch-prediction techniques to predict when a load instruction will miss the last-level cache and applying latency tolerant techniques. Also shown are techniques to improve accuracy by coupling the load-miss predictor with a memory-dependence predictor and pointer-chasing detector.

- **Value-copy-free Latency Tolerance** This dissertation presents an energy-efficient latency tolerance technique which does not store redundant copies of instructions or register data, and does not require expensive data-transfers between slice-buffers, issue-queues, re-order buffer, or register-files.

- **Energy-efficient slice-buffer** This dissertation presents a novel slice-buffer whose instruction contents are executed serially. Knowing that instructions will execute serially allows for techniques to improve energy efficiency. Expensive broadcasts and tag-comparisons are reduced because an instructions dependents are known to be in the slice-buffer, rather than in the issue-queue. Also, physical registers can be shared among instructions because there will be no read-after-write hazards.

1.4 Previously Published Work

This dissertation contains work that was previously published.

- **Flexible Register Management with Reference Counting.** This 2012 work appeared in the 18th International Symposium on High Performance Computer Architecture [4]. It described the reference counting implementation and studies on V_{dd}-gating, register sharing, and execution-driven register reclamation.

- **Register Allocation and V_{dd}-gating Algorithms for Out-of-order Processors.** This 2013 work appeared in the 30th International Conference on Computer Design [5]. It described the register allocation and V_{dd}-gating algorithms building on top of register reference counting.

This dissertation differs from the previous work in the following ways:

- The register reference count and V_{dd}-gating architecture is described with more detail
- The register allocation and V_{dd}-gating mechanisms are evaluated more thoroughly, with detailed synthesized implementations for each algorithm.
- A new V_{dd}-gating technique to keep banks disabled until the first instruction issues is introduced and evaluated.
- A novel predictive latency tolerant architecture is introduced and evaluated, which fundamentally relies on register-reference counting.

1.5 Dissertation Organization

The work presented in this dissertation is organized as follows. First, Chapter 2 presents background on out-of-order architectures, including scaling structures involved in managing the instruction window to tolerate cache-misses. Chapter 3 describes the reference-count register management implementation and introduces several register allocation and power-gating algorithms informed by register-reference counts. This chapter also introduces mechanisms using the microarchitecture and circuit techniques to improve the performance of these algorithms. Chapter 4 evaluates the energy efficiency of these algorithms and compares to earlier power-gating work. Chapter 5 describes SCREW (**SC**aling **R**esources in **E**fficient **W**ays), a predictive and value-copy-free latency tolerant design. This chapter describes the overall architecture, introduces the load-miss predictor and value-copy free in-order queue, and describes several techniques to improve performance of the design. The design is evaluated and compared to existing latency tolerant designs in Chapter 6. Chapter 7 concludes.

Chapter 2: Background and Related Work

Tolerating latency in out-of-order cores and reducing core energy are two focal points of architecture research. This section provides an overview of the canonical structures in out-of-order processors in the context of the instruction window and describes proposed techniques to scale each structure, relating each technique to those proposed in this dissertation. This section also describes several circuit mechanisms for reducing CPU leakage energy, and briefly compares to the approaches proposed in this dissertation.

2.1 Out-of-Order Architectures

Out-of-order scheduling is the fundamental technique allowing processors to tolerate latencies associated with cache misses and long-latency instructions (*e.g.* FPSQRT and FPDIV), exploiting ILP and MLP in the instruction window. Large window out-of-order processors can provide significant performance improvements. Figure 2.1 shows the pipeline for a typical out-of-order processor, with structures defining the instruction window shaded in grey. The front-end from fetch through dispatch stages executes in-order. Instructions are scheduled and executed out-of-order in the middle, but are retired in-order to ensure correct behavior. The following sections describe the various hardware structures shown in Figure 2.1 which allow the processor to perform out-of-order execution and in-order retirement. These include register management structure (register file (RF), free-list and map-tables), issue queue (IQ), load/store queues (LSQ), and the re-order buffer (ROB).

Figure 2.2 shows an execution profile for an in-order processor (a), a large window out-of-order processor (b), and a small-windowed out-of-order processor (c). When a cache-miss or long-latency instruction occurs, the in-order processor (a) must wait until the long-latency operation is completed and the data is available. Execution is serial, and all instructions must wait whether they need the long-latency data or not. Memory instructions cannot be overlapped because the youngest load instruction (7) depends on the outcome of instructions which must wait to execute (5,6), preventing

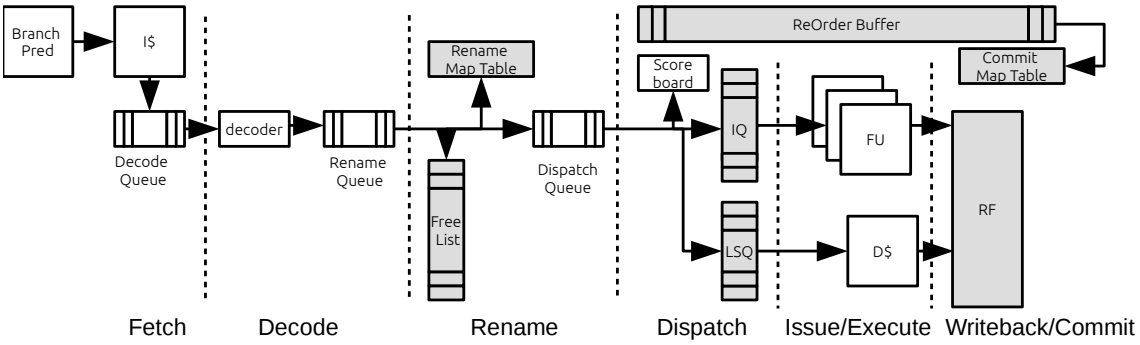


Figure 2.1: Out-of-order processor. Pipeline shows instruction flow from fetch-stage through instruction retirement. Structures defining the instruction-window are shaded grey.

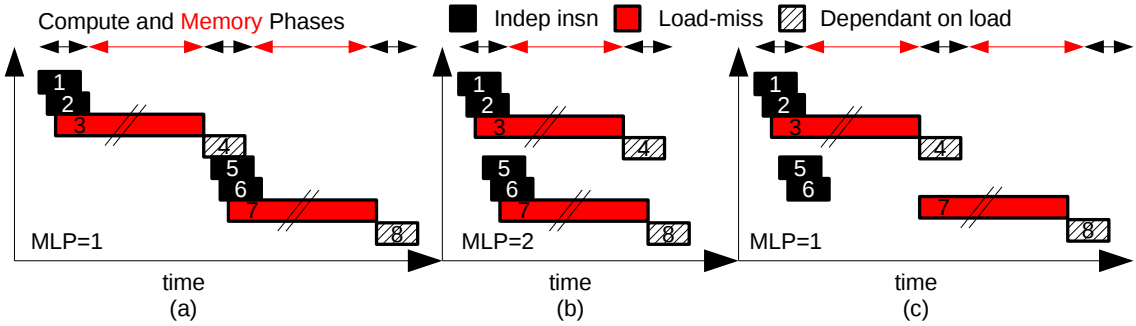


Figure 2.2: Instruction window. (a) serial execution, (b) out-of-order execution with large instruction window able to overlap load misses, (c) out-of-order execution with small instruction window, unable to overlap load misses. Note that the instruction window can change how compute (short latency) and memory (long-latency) phases are overlapped.

memory-level parallelism from being extracted.

An out-of-order processor allows instructions in the window that are independent of the long latency operation to make forward progress. In Figure 2.2 (b), the two long-latency operations (3 and 7) can be overlapped, as the instruction window is large enough to schedule both loads. In this case, MLP is extracted. In (c), the window is too small and the second load (7) is blocked from entering until older instructions retire. Compute instructions (1,2,5,6) are overlapped, but the memory instructions (3 and 7) are not, resulting in execution that is similar to the in-order case.

2.1.1 Register Management

Out-of-order processors rename instruction output registers to remove false dependencies, mapping the small set of architected registers (also called logical registers) to a larger set of physical registers (abbreviated as pregs). The register rename stage is responsible for two tasks: allocating physical registers to instructions, and mapping instruction operand logical register identifiers to physical register identifiers.

There are two approaches to managing the pool of rename registers in out-of-order processors. The first is a ROB based approach used in Intel P6 architectures, where each entry in the ROB contains the destination register value. Instructions update the ROB after the execute and generate a data value. When the instruction retires, the output data is copied from the ROB to the architectural (or retirement) register file [57]. This requires instructions to search both the ROB and the register file to acquire the latest version of their input register operands prior to execution.

The second approach, introduced by the MIPS R10000 [93] is shown in Figure 2.3 and is the mechanism under study in this dissertation. In the MIPS approach, the register file itself contains both in-flight and architected state, indexed by physical register tag, rather than storing data in the ROB. The register file contains many more registers than are listed in the ISA, with map tables defining the relationship between physical register in the register file and architected (logical) register in the ISA. The ROB contains physical register numbers pointing to the registers storing data values rather than the actual data as in the Pentium approach.

In this scheme, a free list is needed to store the un-allocated register tags so that new instructions

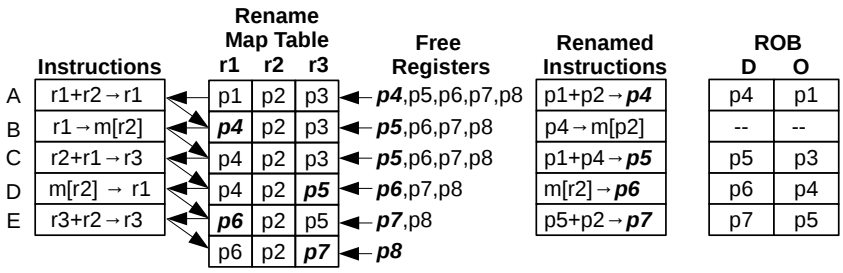


Figure 2.3: Conventional register management and register renaming. Instructions are allocated a destination register from the free list, *e.g.* instruction A is allocated register p4. The rename map table is updated with the new allocation, un-mapping register p1. The ROB entry stores the destination and overwritten (previously mapped) register.

can be given an un-allocated physical register. Typically this is implemented using a FIFO queue. When a younger instruction enters the rename stage, it is assigned a destination register by dequeuing the tag at the head of the queue (shown in *bold italic* in Figure 2.3) and updates the rename map table entry with this new number: *e.g.* instruction A gets preg 'p4' mapped to logical register 'r1' as its destination. This register (p4) is stored in the ROB as the destination register ('D' in the figure). The previous mapping (p1) is stored in the ROB as the overwritten register ('O' in the figure). This register (p1) is freed once the instruction commits and exits the ROB.

2.2 Scaling the Register File

Registers are allocated in program-order at the front-end of the pipeline (the rename stage shown in Figure 2.1). Registers are also freed in program order, when the instruction retires and exits the pipeline. However, register data is consumed within the out-of-order execution region, meaning that there can be a significant delay from when the register was read for the last time and when the register is finally freed. The only other mechanism for reclaiming physical registers is when a branch mis-prediction is detected: instructions executing down the wrong path are squashed and their register destinations are released back to the free list.

These in-order constraints limit the instruction window size, as the window can only support as many register-writing instructions as there are physical registers. A larger instruction window requires a larger register file, increasing area, power, and complexity of the design. The register

file is on the critical execution path, so larger register files can often result in performance loss, due to the increased register access latency. Larger register files are slower due to increased wire capacitance, which can cause pipeline depth to increase as the register access is split over multiple cycles. This not only increases complexity in the instruction scheduler but increases the branch mis-prediction penalty as well. Additionally, the register file is high-bandwidth structure which must serve many instructions in flight at the same time. Wide superscalar processors increase the bandwidth requirements on the RF and requires more read and write ports to service multiple instructions concurrently. An n-wide processor will need at most $2n$ read ports to access input operands and n write ports for its output data. Each additional port increases area and power costs for the register file.

Larger instruction windows place pressure on the register file; a larger amount of instructions in flight will need more registers in which to write their results. One set of techniques logically scale the register file by allocating fewer registers to instructions. This is achieved by sharing registers with common values (such as 0 or 1) among multiple instructions [62, 79]. Another technique to allocate fewer registers is to store results in the rename map table directly [47]. If the value is known to be narrow and only requires fewer than 6-bits rather than the full 64-bit width, then the value can be stored as an 'immediate' in the rename map table itself, rather than consuming a register.

Another set of techniques improve RF efficiency by releasing registers earlier than the commit-stage, freeing the physical register after the last instruction has read the register [82, 35, 50, 19]. This requires additional register management structures to track in-flight consumers of registers and additional locations to store captured register values. In BOLT, the slice-out process (the process by which BOLT tolerates long latency cache misses, described in detail later) requires miss-dependent instructions to "capture" any load miss-independent register inputs. The process requires the register to be read and the value copied into either the ROB or slice-buffer. This requires a wider ROB in BOLT and CFP, where conventional processors would only store narrow register pointers rather than actual values.

Other techniques virtualize the register file, allocating a register only when the instruction is

ready to write the result [51, 13]. However, care must be taken to ensure that physical registers are available for late-allocation to instructions, otherwise the pipeline must stall while waiting for physical registers to be released. This technique also introduces a large virtual-register to physical-register map table, which can be expensive in terms of area and energy.

2.2.1 Scaling the Re-Order Buffer

The re-order buffer (ROB) is a hardware queue which tracks all instructions in flight. Before an instruction can be inserted into the pipeline, it needs a slot in the ROB. Like the free-list, the ROB is a queue with a head and tail pointer. Similar to the register management mechanisms, new instructions are enqueued at the tail of the the ROB in program order and instructions are removed from the head of the ROB in-order. The ROB supports fine-grained recovery from a pipeline flush. For example, when a branch has been mis-predicted, the pipeline will be full of wrong-path instructions. These instructions need to be removed from the pipeline before they perturb the architected state. The ROB entries from the tail-pointer (youngest instruction) up to the branch instruction are invalidated. The branch becomes the new tail position as instructions are fetched from the correct path.

The ROB is designed to enforce program-order instruction completion and to support precise recovery, effectively creating a checkpoint of processor state for each cycle. The requirement to support single-cycle rollback means the processor needs a ROB entry for every instruction in the window. A relaxed roll-back requirement supports processor checkpointing. Instead of a requiring ROB entry for every instruction, the processor is checkpointed periodically. If the processor has mis-speculated or has some other exception and needs to flush the pipeline, the processor recovers to the checkpointed state. This amplifies pressure on other structures in the processor, notably the register file, issue-queue, and load-store queues.

Many latency tolerant techniques such as CFP [82], BOLT [35], Runahead [56], and Waiting Instruction Buffer [45] use speculative retirement to scale the ROB. When a long latency instruction reaches the head of the ROB preventing younger instructions from committing, the processor state is checkpointed, and instructions are removed from the ROB and instructions independent of the load-

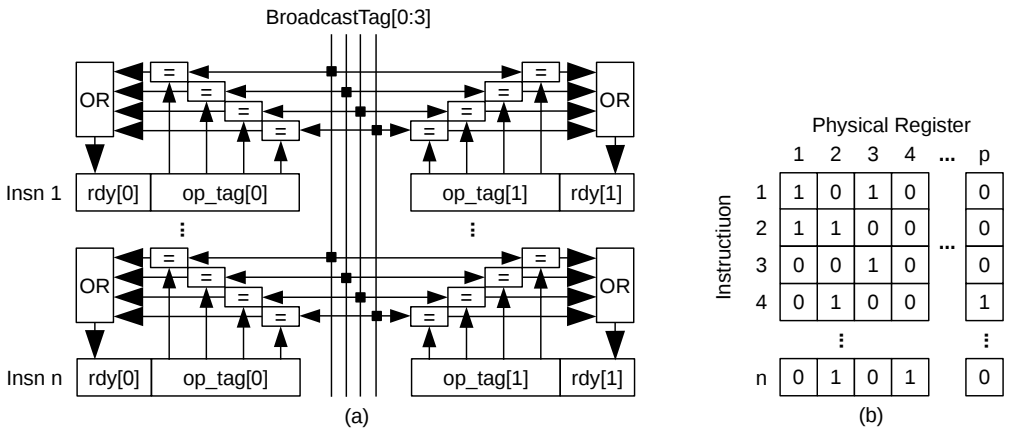


Figure 2.4: Issue Queue organization. (a) CAM based IQ for 4-wide super scalar processor. Up to four instructions broadcast their destination preg tag each cycle. Each destination tag is compared against both source operands for every instruction in the queue. (b) Matrix style structure with 'p' columns—one for each preg in the register file— and one row per instruction. A pending instruction sets the columns indicating the registers it depends on. When an instruction writes to a preg, it clears that column.

miss are able to retire. The limitation to ROB checkpointing is re-execution, recovery overhead, and the additional storage and management of architected state. If the checkpoint distance is very large, then many instructions will be re-executed during a mis-speculation, increasing power consumption. Techniques such as runahead re-execute all younger-than-miss instructions, rather than only those dependent on the miss.

2.2.2 Scaling the Issue Queue

The issue queue holds pending instructions in the instruction window whose source operands are not ready. This set of instructions represents the 'pending' portion of the instruction window, and scaling this pending window can be an effective method for improving ILP extraction. A larger selection of instructions will allow the scheduler to find more independent instructions.

Instruction scheduling typically consists of two stages: wake-up and select. There are two categories of instruction wake-up logic: MIPS R10K style CAM-based logic [93] and matrix-based logic [76]. When an instruction finishes execution, it needs to "wake-up" its dependents and let them know that their source operand is ready. The writing instruction broadcasts its destination physical

register tag to the issue queue, where each instruction in the queue compares the tag against its operands. If the source operand matches the broadcast, it sets a valid bit. When both operands are valid, the instruction is ready to be scheduled. The select stage examines the pool of ready instructions and selects new instructions to issue. Typically this is achieved though a priority encoder, where older instructions are issued in advance of younger instructions.

Instruction queues consume a significant amount of power due to the broadcast and compare operation in the wake-up stage. The delay of the scheduler is proportional to the number of entries in the queue [76]. CAM based wake-up logic is shown in Figure 2.4(a). The IQ CAM must be multi-ported to support the superscalar width of the processor. For an n-wide processor, the CAM must support $2n$ comparisons and n write ports for each source operand [69]. Matrix-style logic, shown in Figure 2.4(b), is limited by the number of entries in the dependency matrix. This matrix is proportional to the size of the IQ and physical register file. Larger IQs and larger register files will have larger and wider dependency matrices to manage, *e.g.* a 36-entry scheduler with 160 physical registers requires a 760B matrix.

Techniques to scale the issue queue include new designs to physically scale the matrix [76] or CAM [9], pipelining the wake-up and select logic[83], or using pointer based wake-up [26, 69]. Other techniques scale the IQ by shifting instructions known to depend on long latency instructions to another buffer, freeing up IQ resources for short-latency instructions. Out-of-Order commit processors [14] move instructions to a "slow lane IQ" buffer if they depend on a long-latency load operation, while BOLT [35] and others [45, 82] pseudo-execute and slice latency-dependent instructions to a slice buffer. These scaling techniques must be performed in concert with techniques to scale the ROB, physical register file and load/store queues, otherwise ILP and MLP will be limited by those other resources constraining the instruction window.

When a miss returns, BOLT and CFP both re-execute the miss-dependent instructions. The instructions are then re-renamed and allocated a destination register, then re-injected into the issue queue where they execute as normal. This re-execution may result in part of the instruction chain slicing out again if a load in the slice misses the cache.

2.2.3 Scaling Load and Store Queues

Latency tolerant (LT) microarchitectures must also scale the load and store queues. Traditional out-of-order processors use CAMs for these queues, which (like the issue queue) are difficult to scale to the sizes required for large window processors. Prior work has shown that speculatively indexed (rather than associative) store queues [78] can scale the store queue to larger sizes. However, physically scaling the queue is not enough. Memory consistency must be maintained when the instruction window contains a mix of speculatively retired and non retired instructions. The "chained store buffer" [33] provides non-speculative forwarding to "deferred" loads which have been sliced out. SVW [72], is extended with some techniques [34] to make it compatible with checkpointed-based management of deferred instructions for a scalable load queue, tagging loads and stores with sequence numbers to make sure that a younger stores do not collide with older loads if they do not execute in order.

2.3 Circuit Techniques for Low Power

The following sections describe circuit techniques which can reduce the energy cost of large structures in the microarchitecture. In this dissertation, we apply low-power circuit techniques opportunistically to reduce the energy cost of large register file SRAM structures.

2.3.1 Power-Gating

Power-gating is a circuit technique that can dramatically reduce leakage energy component by adding a PMOS "gate" transistor between the Vdd power-rail and the logic circuit or an NMOS "gate" between the common ground and the logic circuit [65, 80]. When the "gate" transistor is disabled, the path between the power rail and the logic circuit is left floating. An NMOS power-gating circuit is shown in Figure 2.5. This has two effects on the circuit—first, it interrupts the power-to-ground leakage pathway, introducing a high-resistance node preventing leakage current from passing as easily. The second effect is the reduction in leakage due to the "stacking" effect. Introducing more transistors between the power and ground rails adds more resistance to the power-ground pathway. This is one reason that multi-input logic gates (*e.g.* nand, nor, and-or-invert) have lower

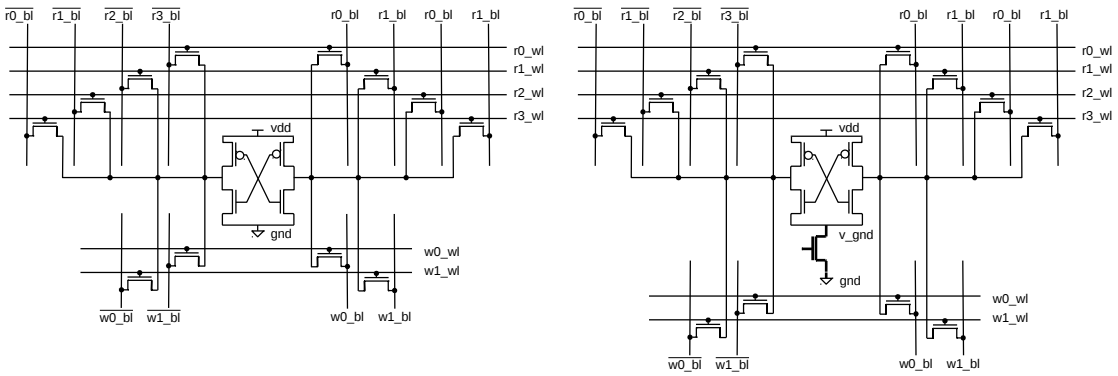


Figure 2.5: Power-gating circuit. The schematic on the left shows a single register file bit cell with two write ports (w0 and w1) and four read ports (r0-r3). The bit cell consists of a cross-coupled inverter, while the I/O ports are pass-gate NMOS transistors controlled by bit- and word-line drivers. On the right, a virtual-ground node is created by adding an NMOS transistor between the inverter NMOS source nodes and ground. When the virtual node is left floating by disabling the gate-transistor, the leakage path is cut and leakage current drops by two orders of magnitude.

per-transistor leakage currents than simple two-transistor inverters.

Power-gating is a destructive operation and will clear any 'state' held in memory cells. When power-gating a memory circuit, only empty cells or cells whose contents are known to be expired and not needed may be power-gated. Power-gating requires a PMOS gate-transistor, driver, and additional isolation circuitry to ensure un-gated logic is unperturbed. The cost to switch these circuits *must* be recovered by the leakage energy reduction in order to break-even and be advantageous compared to clock-gating, which has no intrinsic circuit cost to the RF itself.

2.3.2 Drowsy Caches

Drowsy caches [24] is a technique which targets leakage power in SRAM memories. When the working set for an application fits within the cache, there will be excess capacity that is not used. Leakage power can be reduced by incorporating a multi-level supply-voltage (V_{dd}) for each cache-line. Cache lines that are not frequently re-used can be placed into a low-power drowsy/retention mode, operating at a lower voltage. The drowsy mode supply voltage is lower than the normal V_{dd}level, but not low enough that the cache-line is V_{dd}-gated and loses its data. The cost of this technique is extra wake-up cycles to drive the cache-line to normal supply voltage levels which allows data to be

read from the bit lines. If the drowsy lines are not re-accessed, then there is no performance penalty. When a drowsy cache line is updated with fresh data from memory, the cache line is 'woken' and placed into the normal power mode.

This technique opportunistically scales the cache leakage energy by monitoring cache line use over a large window size. However, this technique does not scale well to register files. A cache will contain hundreds or thousands of cache-lines which are eligible for entering drowsy mode, where the performance penalty of an extra cycle latency has less of an effect on total system performance, especially on lower level (L2, or L3) caches. The register file is on the critical path, and latency must be deterministic for scheduling instructions. Any additional latency will be amplified, as the register file is encountered by nearly all instruction types, not just load instructions.

2.3.3 Power-Gated Register Files

Previous work has studied fine-grained gating of individual SRAM cells in the Register File. Several previous studies have focused on fine-grained gating of individual registers, but without detailed analysis of the energy, performance, or area costs associated with such fine-grained partitioning. Goto and Sato proposed a dynamic gating algorithm using free-list allocation in out-of-order processors, toggling individual registers when they are enqueued and dequeued from the free-list [27]. Khasawneh and Ghose proposed an adaptive technique to disable registers in two places: when the register is allocated but has not been written, and when the register has been both written and consumed but not de-allocated [42]. These techniques focus only on bitcells and do not address the large leakage power consumed by the periphery circuitry within an SRAM register file.

Chapter 3: Flexible Register Management

Conventional out-of-order processors use a unified physical register file to store values produced by in-flight instructions and committed instructions. These processors allocate and reclaim registers—for the rest of this dissertation we use *register* to mean the 64-bit physical register—using a circular queue free list. This chapter presents a detailed analysis of an alternative and more flexible register management scheme—reference counting. We describe reference counting designs that support micro-architectural techniques including register file power gating, dynamic register move elimination, register file checkpointing, and latency tolerant execution. We present detailed performance and circuit simulations which show that the energy cost of reference counting is low, and easily recouped by the savings of the techniques it enables.

3.1 Conventional Register Management

Most contemporary out-of-order processors use a unified physical register file to hold both architectural register state from committed instructions and speculative register state from in-flight instructions. This design requires fewer register value copies than one that uses a separate architectural register file and a ROB that holds destination register values. Processors manage these physical registers using a variant of the MIPS R10000 register management algorithm [93] with a free list that operates as a circular queue.

The free list operates as a circular queue FIFO with head and tail pointers. At rename, a register-writing instruction is allocated a destination register 'D' from the head of the free list. The instructions dequeue the register from the head of the free list and update the rename map table, remapping the logical register to the dequeued value and un-mapping the previous physical register. A map table look-up records the register previously allocated to the instruction's logical destination register—this is the overwritten physical register O. Both destination 'D' and overwritten registers 'O' are recorded in the instruction's ROB entry. When an instruction commits, overwritten register

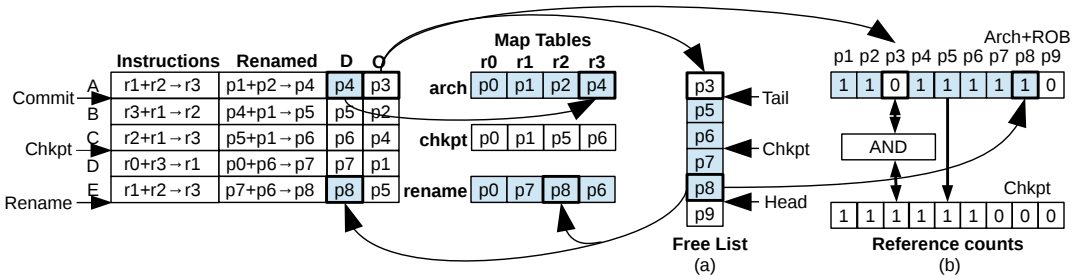


Figure 3.1: Register management example. (a) shows the circular FIFO free list used in conventional processors containing $P - L$ entries, with head, tail, and checkpoint pointers. (b) shows a bitvector reference count formulation of the free list and checkpointed state. Five instructions in the ROB (A–E) show two actions: Instruction A commits its result, adding its register to the architectural map table and enqueuing its overwritten preg to the tail of the freelist (a) or clearing the bit in the reference count vector (b). Instruction E renames and dispatches, updating the rename map table and dequeuing register p8 from the head of the free list (a) or setting the bit in the reference count vector (b).

O is freed and added to the free list at the tail. When an instruction is squashed, destination register D is freed and added to the free list at the head—this action only moves the free list head pointer.

The rename map table is a multi-ported RAM containing (physical) register numbers indexed by logical register number. For an architecture with L logical registers defined in the ISA and a register file with P physical registers, the rename map table will have L entries storing a preg number that is $\log_2$P-bits wide. The free list is a RAM of physical register numbers managed as a circular queue. For P registers and L logical registers, the free list contains $P - L$ entries as L registers are always mapped. Each entry is $\log_2$P-bits wide. The head and tail pointers are $\log_2\ P - L$-bits wide and are incremented using a $P - L$-bit counter.

Figure 3.1 shows an abstract processor with ten physical registers (p0–p9), four logical registers (r0–r3), and a six-slot free list ($P - L = 6$). Logical register r0 is hardwired to the value zero and mapped to hardwired "register" p0. The figure shows the architectural/commit map table (AMT), rename (speculative) map table (RMT) and five instructions (A–E) in the ROB. Each instruction is shown in raw (logical register) and renamed (physical register) form along with its destination (D) and overwritten (O) registers. The figure shows two actions: instruction A commits—it writes its *destination* register (p4) to the architectural map table and frees its *overwritten* register (p3) by

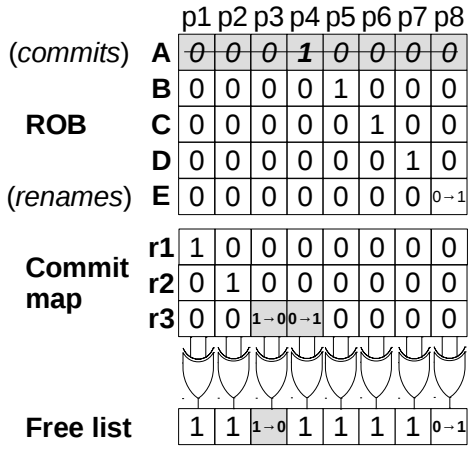


Figure 3.2: Reference count matrix. Each column in the matrix represents an entity that can hold a register tag. In a conventional processor, this is the ROB and commit map table. Five instructions are in the ROB (A–E) with unary matrix register management. Instruction A commits, exiting the ROB and affecting all bits shown in grey. It updates the commit-map table, incrementing row r3, column p4 and decrementing row r3, column p3. Register p3 is now free in the free list vector (bit=0). Instruction E renames its destination register and increments p8, which is shown as allocated (bit=1) in the free list vector

writing it to the tail of the free list. Instruction E renames and dispatches—it allocates destination register p8 from the head of the free list and writes that register number into the rename map table. Also shown in Figure 3.1, a checkpoint is created at instruction C, between instructions A and E. Checkpoints stores the committed register mapping and free-list head pointer, allowing the processor recover from branch mis-predictions and exceptions [38].

3.2 Register Reference Counting

The first description of register reference counting used per-register multi-input up-down counters [53]. A per-register counter keeps a count of issued but un-executed instructions in the processor pipeline that read from physical register. An instruction that reads a register increments that preg's counter as soon as its inputs are mapped from logical to physical registers. The instruction decrements the counter after it has read the value. A physical register can be reused by a new instruction when it has been written to, un-mapped from the rename map table, and its counter is zero. The total number of issued but un-executed instructions in the processor is bounded by the

instruction-window size, so the counter does not have to be very wide. However, even with "small" counters, the required storage can be costly if the physical register file is very large.

3.2.1 Matrix Reference Counting

More recent designs use a two-dimensional unary bit-matrix [73, 79] rather than numerical counters, shown in Figure 3.2. The matrix has a column per physical register and a row per entity that can hold a register, *e.g.*, an in-flight instruction or the commit map table. This structure is similar to the register renaming and map table checkpointing mechanism used in the Alpha 21264 [41]. In this example, instruction A commits and operates on the destination register (p4) and overwritten register (p3) bits in the matrix. Register p4 is no longer referenced in the ROB, but the commit map table. Instruction E is renamed and enters the ROB, allocated register p8. The bits in each column are OR'ed together to create a free list bitvector from which registers are allocated for new instructions.

3.2.2 Vector Reference Counting

In this dissertation, we make use of matrix reference counting. However, when applied to a conventional out-of-order processor, a single-row matrix (or vector) is required. Figure 3.1b shows a compressed vector representation of the reference counts. Instead of a matrix representation with a row for each in-flight instruction, we use a single vector with one bit per physical register to represent all speculative ROB instructions. Register p0 is not reference counted because it represents a hardwired value of '0'. This register does not need to be counted as it is not allocated or freed. The vector has a 1 in position p if register p is mapped to either an architectural register or the destination of an instruction in the ROB. An instruction with destination register D and overwritten register O sets bit D when at dispatch and clears bit O when at commit. If the instruction is squashed due a branch mis-prediction or exception, it clears bit D. A checkpoint can be created by copying the value of the bitvector. The free list then becomes the logical AND of the reference count vector and the checkpoint vector (or vectors), as any checkpointed register cannot be used until the checkpoint is released. We show later that we can support matrix-like operations on rows

of bitvectors—where a row can represent the state of a thread in an SMT processor or indicate the register is shared in a processor such as NoSQ [79]

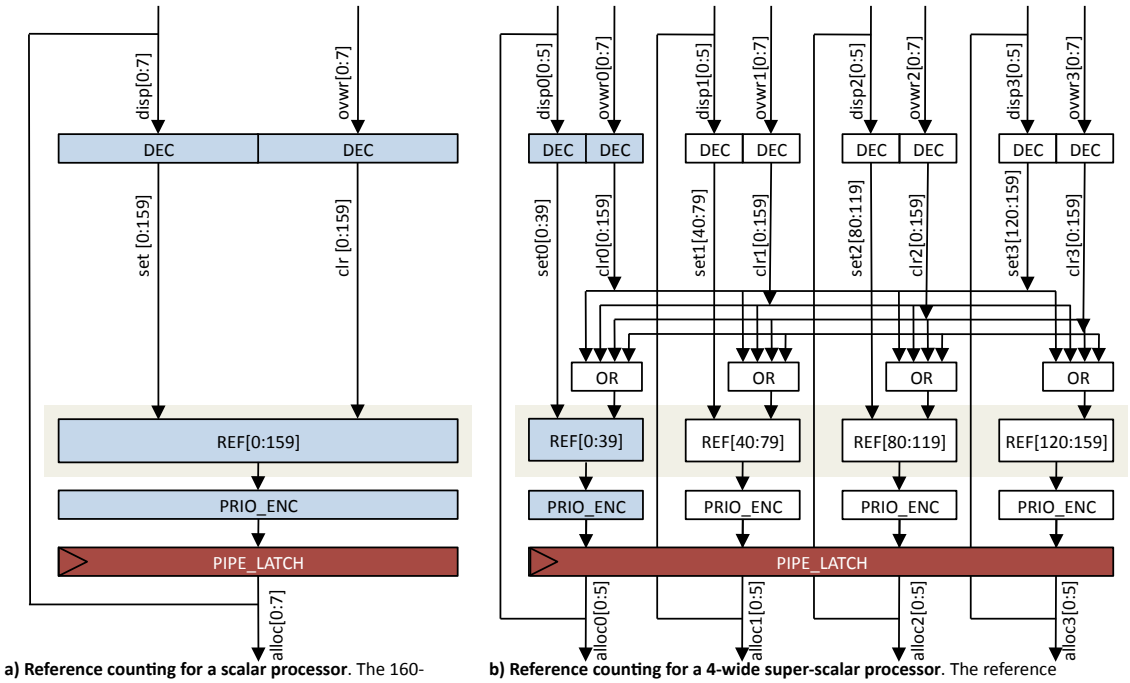
3.3 Implementation

Reference counting hardware can be implemented in a few different ways. This dissertation describes our previously published implementation [4] in more detail. The canonical components required for reference counting are: (i) a vector of state-elements to hold the reference count. This can be latches, flip-flops, or an SRAM. (ii) A set of decoders converting the physical register tag for the allocated and freed registers to a bitvector representation. (iii) an encoder to pre-select a free register for allocating to the next instruction. There is flexibility in the design of (i) and (iii), as different state elements will be appropriate depending on the size of the bitvector and the number of vectors. Flexibility in the encoder allows for different allocation algorithms to be used, giving the designer the opportunity to steer registers to particular portions of the register file. In this section, we describe a priority encoding scheme, where the first empty register is pre-selected. Other allocation schemes are discussed in Chapter 4.

3.3.1 Scalar Implementation

A scalar processor allocates and frees at most one register every cycle, so a traditional free list needs one read port for allocation and one write port for reclamation. Reference count bitvectors support set/clear operations rather than read/write operations and so we implement the bits using set-reset (S-R) latches and the set/reset "ports" using encoders and decoders. A scalar processor with P physical registers uses two P-wide decoders to convert the physical register number to a one-hot representation. This one-hot vector connects to the set or reset input of the S-R latch to either set and clear one bit per cycle. The allocator is a P-bit wide priority encoder. Figure 3.3(a) shows this design for a processor with 160 registers.

Because reference counting bits are implemented using latches and not flip-flops, the identification of free registers and the bitvector updates to indicate that the registers are no longer free cannot take place in the same pipeline cycle. We avoid a cycle by "pre-selecting" a free register one pipeline



a) Reference counting for a scalar processor. The 160-register reference count vector supports one bit set (from dispatch) and one bit clear (from commit/overwrite) per cycle. Registers are allocated using a priority encoder. The allocated register is the dispatched destination register.

b) Reference counting for a 4-wide super-scalar processor. The reference counting vector is divided into 4 disjoint segments. 4 ¼ width priority encoders allocate 4 registers, 1 from each segment, per cycle. Because the dispatched destination registers are the allocated registers, those decoders are ¼ wide as well. The 4 decoders for the overwritten registers are full width.

Figure 3.3: Reference Counting Mechanism. Processor with P=160 registers (a) scalar processor. (b) 4-way superscalar processor with partitioned bitvectors, decoders, and encoders (N=4)

stage (cycle) ahead. If the register is not allocated in the next cycle, the corresponding bit is not set and the register remains free. The pre-selected value will not change unless a higher priority register is reclaimed.

3.3.2 Superscalar Implementation

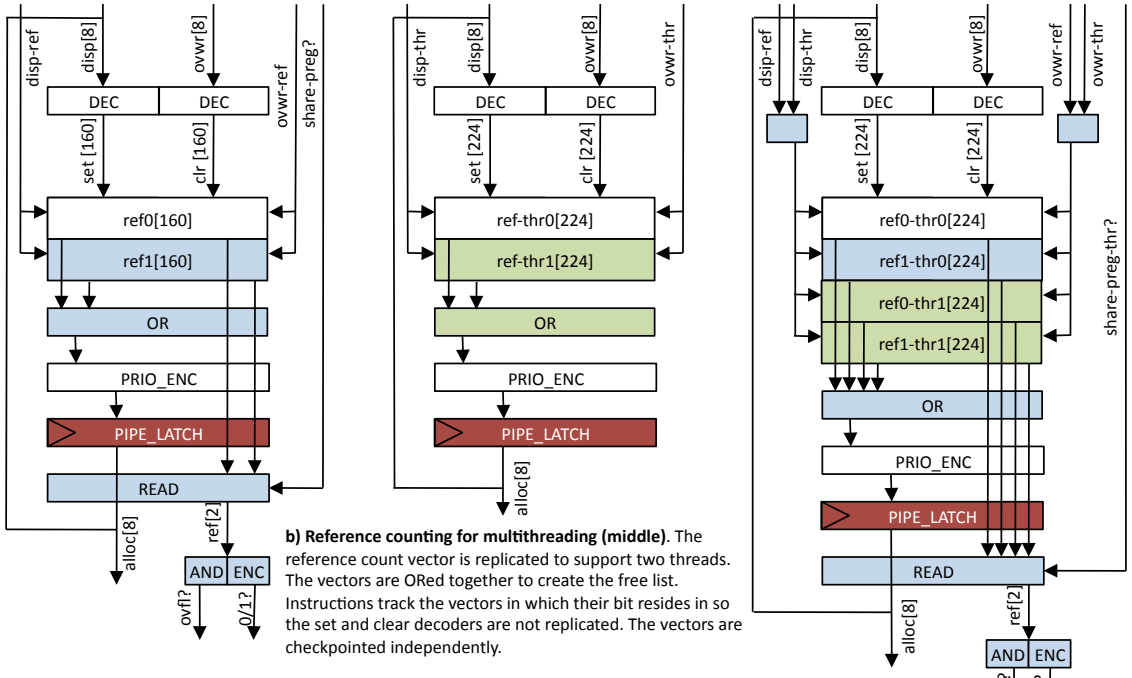
An N-wide superscalar processor can dispatch and commit up to N instructions per cycle. As each instruction can write to a destination register, the processor must be able to allocate and free up to N registers per cycle. Rather than supporting N-wide operation with N read and N write ports, a traditional free list FIFO is organized with one read port, one write port, and N entries per row. The queue discipline makes this organization conflict free, while latching recent results reduces accesses and facilitates management.

A superscalar processor with reference counting must allocate N registers per cycle and also support setting and clearing up to N bits per cycle. To support N-wide allocation, we divide the register space into N sets and make each encoder responsible for only P/N registers, *i.e.*, the vector is only P/N-bits wide, rather than P-bits wide in the scalar case. The decoders that set bits corresponding to the destinations of dispatched instructions are divided in a similar way. Figure 3.3(b) shows reference counting for a 4-wide superscalar processor.

However, this does not mean that supserscalar way-N only allocates from set-N. The assignment of sets to superscalar renaming slots is rotated on a per cycle basis to avoid stalling dispatch when set 0 is empty. Dispatch occurs in parallel, with up to N instructions requesting a register allocation. If the renaming slots were fixed, set 0 would quickly empty—the superscalar width is only an upper bound, not a constant. If the dispatch width is not fully used, the lower order sets in a fixed-mapping will be allocated from more frequently. The decoders that clear bits corresponding to registers overwritten at commit cannot be divided without inducing conflicts—as registers are not overwritten in allocation order—and are replicated at full P bit-width. Registers are overwritten in commit-order in conventional processors, but the overwrite may not correspond to the partitioned bank mapping at the commit cycle—*e.g.*, an instruction could overwrite a register that was in bank 2, but at commit, this register is now mapped in bank 0. a similar constraint exists execution-driven reclamation schemes [35, 79].

3.3.3 Multi-threaded Support

Many contemporary out-of-order processors employ SMT (Simultaneous Multi-Threading) [87, 44] to improve performance. Instructions are scheduled from multiple threads to make sure functional units are continually being used. The instruction window in an SMT processor manages instructions from N different threads at once. Each thread must have its own consistent view of the architecture. This requires replication of certain resources for each thread. For example, each thread must have its own set of architected registers—e.g. an SMT-2 machine will require 128 architected registers, 64 for each thread. This can reduce the number of rename registers available to each thread unless the physical register file size is increased. Each thread also needs its own map table, to identify



b) Reference counting for multithreading (middle). The reference count vector is replicated to support two threads. The vectors are ORed together to create the free list. Instructions track the vectors in which their bit resides in so the set and clear decoders are not replicated. The vectors are checkpointed independently.

a) Reference counting for register sharing (left). The 160-register reference count vector is replicated to allow up to two instructions to share a given register. The free list is created by ORing the two vectors together. Instructions track which of the two vectors their bit resides in and so the set and clear decoders are not replicated. The read port reads the two reference count bits of the requested register, allowing the the processor to test for sharing overflow and to determine—if sharing is allowed—which reference count vector the current instruction should set or clear. Although not shown, both vectors are checkpointed and restored in unison so the reference counting checkpoint table is widened to 320 bits.

c) Reference counting for register sharing and multithreading together (right). The reference count vector is replicated four times to allow up to two instructions to share a given register and two threads. The free list is created by ORing the four vectors together. Instructions track which thread they belong to and which sharing vector within each thread they use so the set and clear decoders are not replicated. Within each thread, the two sharing vectors are checkpointed and restored in unison so the reference counting checkpoint table is widened to 320 bits.

Figure 3.4: Reference Counting Mechanism for SMT.

the physical register which corresponds to a logical register. Extending reference counting for SMT requires replicating the reference count bitvector on a per thread basis. Instructions update bits in the vector corresponding to their thread. Thread reference counts are maintained in separate vectors so that they can be checkpointed separately. However, the reference count checkpoint RAM is shared among the threads. Figure 3.4b shows reference counting support for an SMT processor with two threads. An additional 64 registers hold the architected state of a second thread.

Reference counting supports dynamic register file partitioning while preserving each thread's ability to commit instructions independently of the other threads and to recover to a checkpoint in a single cycle. Supporting this combination of features with a conventional free list requires $T+1$ free lists, each of length $P-(L\times T)$ where T is the number of threads. Even here, only registers freed

by overwriting are shared dynamically. Registers freed by squashing remain with the thread that initially allocated them. A cheaper form of partitioning based on traditional free lists is quasi-static partitioning in which registers are divided statically among *active* threads. For instance, a dual-threaded processor with quasi-static partitioning can allocate all physical registers to one thread or give two threads half the registers each. Quasi-static partitioning can be implemented using T free lists each with P/T registers. When the number of active threads is fewer than maximum, consecutive free lists are treated as a single circular free list. To support these modes using reference counts instead only requires a mask on the reference count allocation vector.

3.3.4 Register Sharing

With conventional renaming mechanisms in an out-of-order processor, every dispatching instruction is allocated a *unique* register target for its output data. If at rename it is known that the value the instruction will produce already exists (or will exist) in some register, then the map table entry corresponding to the instruction's destination can be set to the register that contains (or will contain) this value, sharing the register among multiple instructions. Renaming will naturally route the instruction's dependents to this older source. Out-of-order issue will naturally wake them up as needed. Register sharing effectively removes instructions from the dynamic dataflow graph, reducing its height. It also reduces activity in the out-of-order execution engine. The difficult aspect of register sharing is identifying sharing opportunities, such as register moves and instructions that store the value zero. Register sharing is a mechanism which is supported by reference counting with little extra cost. The benefit is significant, as the occupancy of the register file can be reduced by allocating *fewer* registers to in flight instructions.

Physical register sharing is difficult to implement without reference counting. The reason is that responsibility for freeing the shared register has to be transferred from the instruction that overwrites the older sharer to the instruction that overwrites the younger one. This transfer is difficult to orchestrate. It may also have to be undone if the younger sharer is squashed. Reference counting simplifies the book-keeping by tracking register occupancy in a central location and eliminating the need to assign register freeing responsibility to any specific instruction *a priori*.

Figure 3.4a shows reference counting support a processor supporting register sharing. An additional bitvector allows the register to be shared between up to two instructions. If the rename logic determines that the value is to be shared, it detects if the register is shareable and sets a bit in the second bitvector. The rename map table is updated with an extra bit to allow the system to know which allocation has set vector[0] or vector[1]. The reference count vectors are bitwise 'OR-ed' together, such that registers can only be allocated to a new instruction if they are not present in *either* vector. An extra vector read port checks the state a particular column in the vector to determine if the target physical register is shareable. If the vector has already been shared, then a new register must be allocated for the instruction. This mechanism allows simple register-register move instructions to be eliminated from the instruction stream, as the map-table pointer update performs the equivalent function. Rather than move the contents from register[A] to register[B], register[A] is shared among the older instruction and the move instruction. The map table will point the new logical register to the same physical register.

3.3.5 Squash Recovery with Checkpoints

The MIPS R10000 supported fast single-cycle recovery to a small number of checkpoints—snapshots of the rename map table that can be created and restored in a single cycle but do not support incremental updates [93]. Checkpoint recovery frees an arbitrary number of in-flight destination registers in one cycle by squashing speculative (uncommitted) instructions that entered the instruction window after the checkpoint was taken. The free list supports high-bandwidth squash-triggered reclamation by checkpointing and restoring the head pointer. The tail pointer is only adjusted at commit, and by definition is not affected by checkpoint recovery as committed instructions are not speculative. Register checkpointing is a key requirement for high-performance, latency tolerant microarchitectures and is easily supported by register reference counting. The checkpoint becomes another row in the reference count matrix. Physical registers which are allocated in the checkpoint are not available for allocation to new in-flight instructions.

Figure 3.5 shows a checkpoint taken after instruction C was renamed. Reference counting bulk register release is implemented by adding a P-column C-row RAM, where C is the number of check-

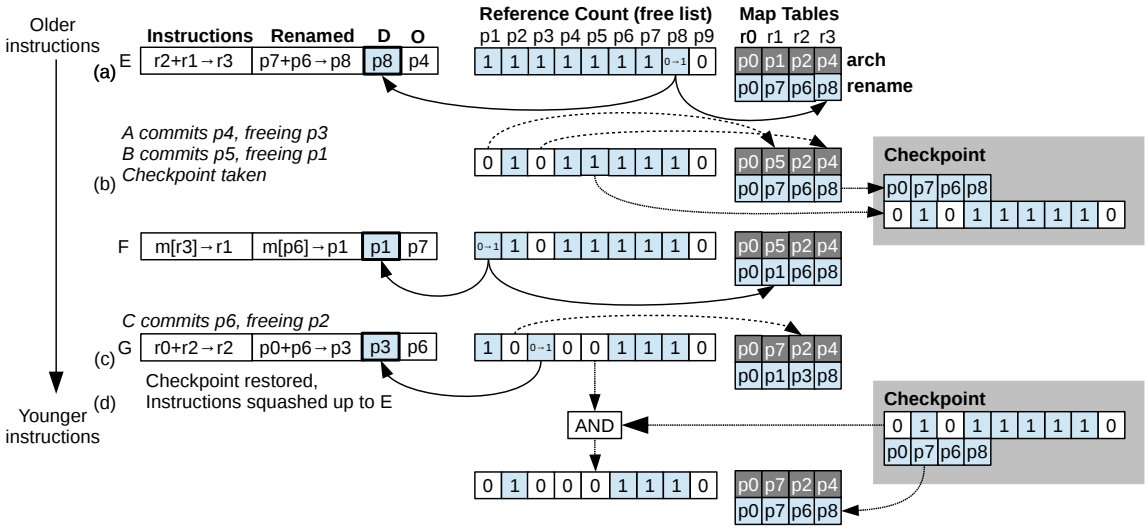


Figure 3.5: Reference Count Checkpoint and Recovery. (a) Instruction E is renamed and allocated register p8. (b) A checkpoint is taken, copying the current rename map table and reference count bitvector. (c) the map table and bitvectors are manipulated by new instructions allocating and old instructions committing. (d) The checkpoint is restored, squashing instructions younger than the checkpoint. The checkpointed rename map table and reference count vector are restored. The bitvector is 'AND-ed' with the current vector to account for registers freed by retirement of instructions older than the checkpoint.

points. When a checkpoint is created, the contents of the reference count vector are written into the checkpoint table in the corresponding row. When a checkpoint is restored, the checkpoint RAM is read and its contents are AND-ed with those of the current reference count bitvector. Figure 3.5b shows this action.

3.3.6 Enabling Speculative Retirement

Several recently proposed high-performance microarchitectures [14, 2, 82, 45] rely on speculative retirement, *i.e.*, the ability to execute, complete, and release the pipeline resources of large numbers of instructions, and then either commit or abort their effects in bulk. The register state of speculatively retired instructions is collapsed into a checkpoint created at the start of (speculative) retirement.

Implementing speculative retirement in a processor with a unified register file requires a retirement map table. It is this map table—not the rename map table—whose contents are checkpointed and restored to support checkpoint/abort operations. This mechanism requires that the processor to either checkpoint the contents of the register file itself or to "pin down" the registers named in

the retirement map table, temporarily removing them from circulation. With a conventional free list, checkpointing register file values is the only option. Value checkpointing is implemented with low area and latency overheads and moderate power overhead using "shadow bit cells" [20]. This approach allows a checkpointed thread to use the full complement of rename registers. In an SMT processor, tracking thread register usage in a bitvector—a simple form of reference counting—allows threads to checkpoint independently using a single set of shadow bit cells.

Reference counting supports the pinning approach. Register pinning does not add area, power, or latency to the register file, but reduces the number of rename registers a checkpointed thread can use. This is not as bad as it first appears because only registers mapped to logical registers that are redefined are lost. Redefinition of all logical registers in a one speculative episode is unlikely, *e.g.*, an integer program is unlikely to redefine many floating point registers.

The reference count bitvector is replicated to support register pinning—one replica tracks register usage by the ROB, the second tracks usage by the retirement map table (ARCH). These two vectors are manipulated using three decoders. One decoder sets the bit for the dispatched destination register in the ROB vector. A second decoder clears the bit for the overwritten register in the ARCH vector. The third decoder "moves" the bit for the destination register of the committing instruction from the ROB vector to the ARCH vector. Speculative retirement checkpoints can share the RAM used to hold checkpoints for single-cycle branch misprediction recovery. We make use of reference counting and speculative retirement in Chapter 5, in a new latency tolerant microarchitecture.

3.4 Implementation Costs

We present the first detailed measurement of an efficient reference count implementation of a register management scheme supporting speculative retirement. We evaluate the cost of replacing conventional register management components with a reference count approach by building each component in RTL and measuring the energy cost. We build circuit models of free lists, register files, and checkpoint SRAMs using the NCSU FabScalar memory generator [12] in NSCU's 45nm FreePDK CMOS technology. The reference count hardware is synthesized using the Synopsys RTL tool-chain with the NCSU 45nm technology. We generate activity factors for each block by measuring accesses to

Table 3.1: Register management area and power for single- and multi-threaded four-way superscalar processors.

Component	μm^2	mW(dy)	mW(st)	mW(av)
Single-threaded				
Register file, 160×64b, 6r3w	109690	22.70	2.49	9.97
Freelist, 27×32b, 1r1w	2279	2.08	0.06	0.82
Register file + freelist	111012	**24.78**	**2.55**	**10.79**
Refcount, 1×160b, 4r4w	7622	1.24	0.08	0.72
Checkpoint, 4×160b, 1rw	2639	1.33	0.06	0.19
Register file + refcount	118994	**25.27**	**2.63**	**10.88**
Multi-threaded				
Register file, 224×64b, 6r3w	152226	24.50	3.49	11.00
Freelist, 27×32b, 1r1w	2279	2.08	0.06	0.82
Register file + freelist	155190	**27.45**	**3.56**	**11.82**
Refcount, 2×224b, 4r4w	14515	1.54	0.14	1.00
Checkpoint, 4×224b, 1rw	3658	1.91	0.08	0.27
Register file + refcount	170399	**27.79**	**3.71**	**12.27**

Table 3.2: Register management area and power for register sharing.

Component	μm^2	mW(dy)	mW(st)	mW(av)
No move elimination				
Register file, 160×64b, 6r3w	111320	22.70	2.55	9.68
Refcount, 1×160b, 4r4w	7574	1.24	0.08	0.72
Checkpoint, 4×160b, 1rw	2639	1.33	0.06	0.19
Register file + refcount	**121533**	**25.27**	**2.69**	**10.59**
Move elimination				
Register file, 160×64b, 6r3w	111320	22.70	2.55	9.25
Refcount, 2×160b, 4r4w	12317	1.32	0.12	0.84
Checkpoint, 4×320b, 1rw	5186	1.80	0.11	0.29
Register file + refcount	**128823**	**25.82**	**2.75**	**10.38**

each component using a cycle-level performance simulator. This simulator executes user level x86_64 code and is modeled roughly after Intel's Nehalem microarchitecture [74]. It is 4-wide issue with a 23-stage pipeline, 128-entry reorder buffer, 36-entry issue queue, and 96 rename registers. We model a single-threaded configuration with a register-file containing 160 registers (96 rename + 64 architected). Further details of the benchmarks and simulator are described later.

Table 3.1 compares the power and area of the free list and reference counting designs for single- and dual-threaded processors. We use a register file design with 6 read ports and 3 write ports, typical for a 4-wide issue processor. In the table, dynamic power assumes maximum activity whereas

Table 3.3: Register management area and power for speculative retirement.

Component	μm^2	mW(dy)	mW(st)	mW(av)
Single-threaded				
Register file-sb, 160×64b, 6r3w	130828	24.40	3.00	11.30
Freelist, 27×32b, 1r1w	2279	2.08	0.06	0.82
Register file-sb + freelist	**133107**	**26.48**	**3.06**	12.12
Register file, 160×64b, 6r3w	109690	22.7	2.49	9.97
Refcount, 2×160b, 4r4w4rw	**10641**	**1.38**	**0.11**	**0.86**
Checkpoint, 4×160b, 1rw	2639	1.33	0.06	0.19
Register file + refcount	**130828**	**25.41**	**2.66**	**11.02**
Multi-threaded				
Register file-sb, 224×64b, 6r3w	185900	26.60	4.27	12.10
Freelist, 27×32b, 1r1w	2279	2.08	0.06	0.82
Register file-sb + freelist	**188179**	**28.68**	**4.33**	**12.92**
Register file, 224×64b, 6r3w	152226	24.50	3.49	11.00
Refcount, 4×224b, 4r4w4rw	23373	2.54	0.22	1.68
Checkpoint, 4×224b, 1rw	3658	1.91	0.08	0.27
Register file + freelist	**179257**	**28.95**	**3.71**	**12.95**

average power is weighted by dynamic activity factors, collected from the SPEC benchmarks, and incorporates leakage power. For instance, the reference counting checkpoint RAM is a wide, high-powered structure but is written only once every ten cycles on average and read less frequently than that.

Our free list has 108 total entries—more than the 96 rename registers our simulated core has—to support simple register sharing [84] described in Section 3.3.4. For the single-threaded core, reference counting occupies 3 times more area and consumes 11% more power than a conventional free list, although the marginal overhead is small relative to the area and power consumption of the register file—7% and 1%, respectively. In the single-threaded configuration, reference counting and a conventional free list have identical performance.

For the multi-threaded core, we retain the 108 entry free list and add a set of head/tail pointers to statically partition it—and consequently register file–between the two threads. This setup also allows all 96 rename registers to be used by a single thread when one thread is quiesced. Dynamically partitioning the register file between two threads requires a more complicated setup—3 free lists with 108 slots each. With reference counting, multi-threading requires additional hardware over a single-threaded design, both an additional vector and more bits per vector. In a multi-threaded processor,

reference counting consumes 54% more power than a free list—4% of register file power. However, reference counting implements dynamic register partitioning which can improve throughput by 1%.

Chapter 4: Register File Power-Gating

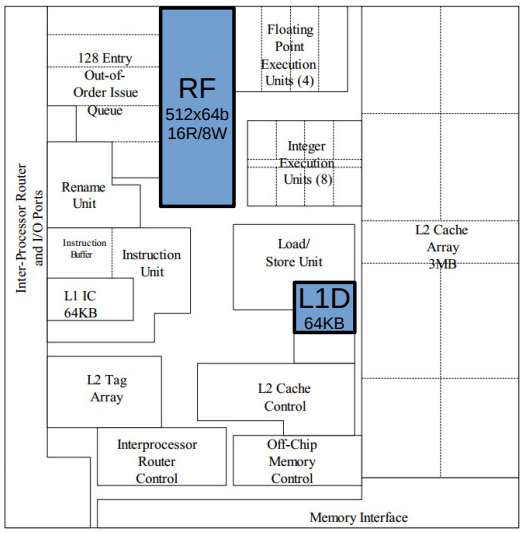


Figure 4.1: Alpha EV8 floor plan. 4KB register file consumes approximately 5× the area as the 64KB L1-data cache.

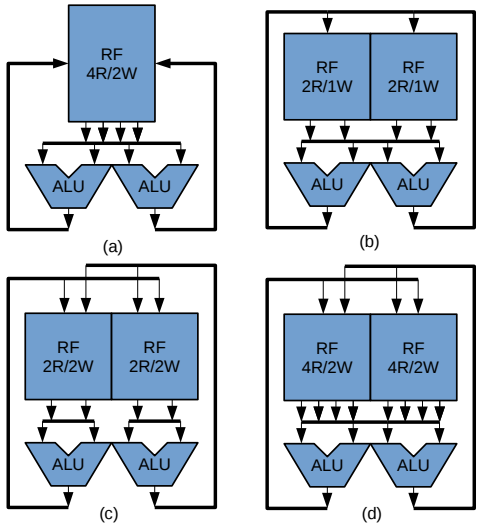


Figure 4.2: Banked register file. (a) monolithic, (b) banked with shared writes, (c) bank-per-way (d) banked for clock-gating

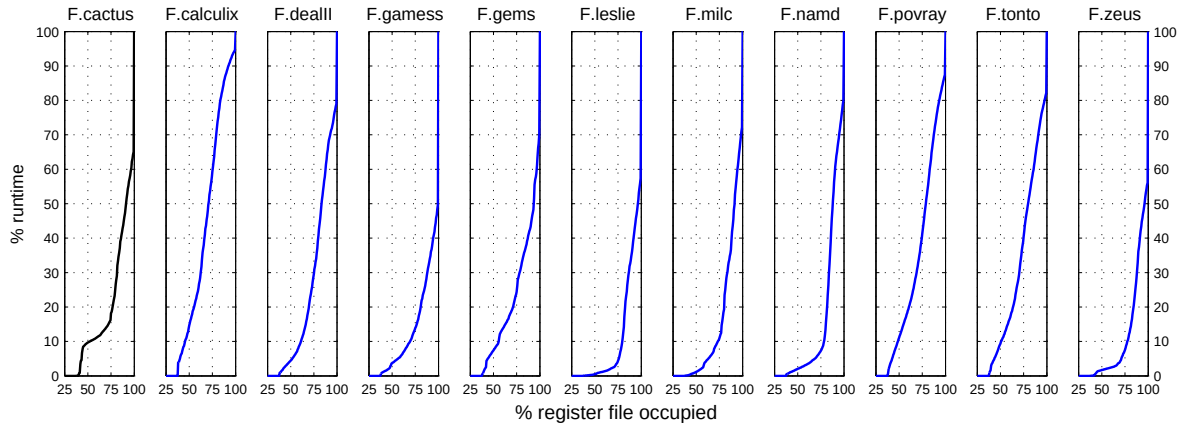
Despite active research in processor power management, a significant portion of a processors active and static power is consumed by register files. This power cost is exacerbated in modern out-of-order processors that use MIPSR10K-based physical register-renaming, as a larger instruction window requires a bigger pool of rename registers, which in turn necessitates a larger register file. The register files in the IBM POWER7 processor from 2010 consume 21% of core power [96], while the Intel Westmere Core i7 register files account for 30% of core power [17]. Furthermore, supporting SMT consumes additional register file space to store the architected register state for each thread. The trend towards cores supporting higher degrees of multi-threading with more thread contexts—from two threads per core in IBM's POWER5 processor [40], introduced in 2004, to eight threads per core in the POWER8 processor [25], introduced in 2014—only exacerbates the pressure on the register file, requiring more register file capacity to store each threads register state. Notably, the eight-threaded Alpha EV8 processor required 256 registers (8 threads × 32 registers) just to store the architectural context for each thread [66]. Due to the high degree of multi-threading and support

for eight-way superscalar execution, the EV8 register file—highlighted in Figure 4.1—was designed with 256 rename registers for a total of 512 registers. This register file required five-times the area of the 64KB L1 data cache [66]!

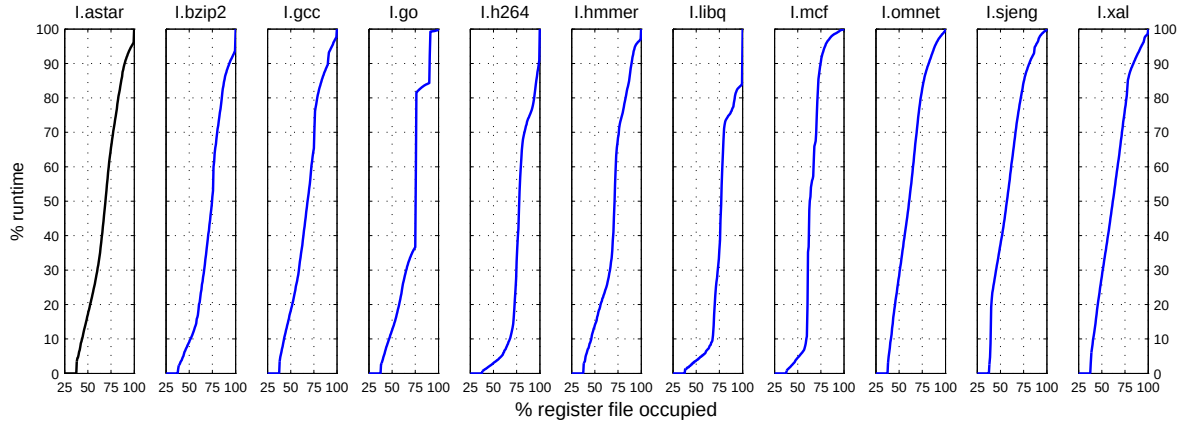
Register file banking is a technique often used to partition the register file into smaller blocks to improve power and performance [28, 86]. Banking the register file chops up the SRAM bitlines into shorter, lower-capacitance segments which require smaller drivers which reduces both access latency and access energy. Clock-gating is often applied to register file banks in order to reduce dynamic power consumption [55, 63]. When clock-gating, data and clock inputs are disabled (gated), and the isolated circuit experiences no dynamic power dissipation, only static dissipation. In this section, this dissertation shows how leakage power can be significantly reduced by using reference counting to power-gate (also called Vdd-gate) register file banks.

Because register file banking has so many advantages, there have been several approaches to partitioning the register file, shown in Figure 4.2b—d. Each technique has its own costs and benefits. In (b), the register file is split into two banks with half as many read and write ports as the monolithic register file (a). In this case, reducing the bank size from the monolithic case reduces latency and energy costs. Reducing the port density improves this reduction by lowering the area cost, as each bit cell now has fewer ports. However, this banking approach introduces a bottleneck which can reduce performance or otherwise offset the energy gains. Bottlenecks are manifested in two ways: when the number of operands required from a single bank exceeds that banks read-port bandwidth, and when results from both ALUs are written to the same bank, exceeding the write-port bandwidth. Reducing this bottleneck requires complex bypass logic, additional buffers, or an extra pipeline stage [86], which attenuates the power reduction.

Figure 4.2c shows the approach used in the Alpha EV8. Here each register file bank is clustered with specific functional units in the processor. This reduces read-port pressure, as only a fraction of the superscalar width (*i.e.* issued instructions) will read out of any single bank. However, this approach requires duplicate entries in each register file bank to ensure that an instruction in one cluster can access data generated from the other. Every register-write is broadcast to both RF banks,



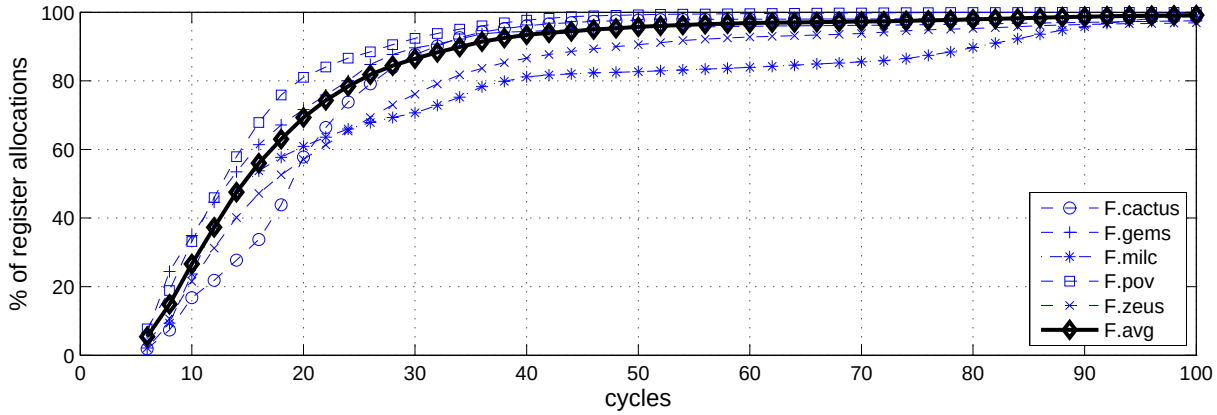
(a) SPEC2006 Floating Point



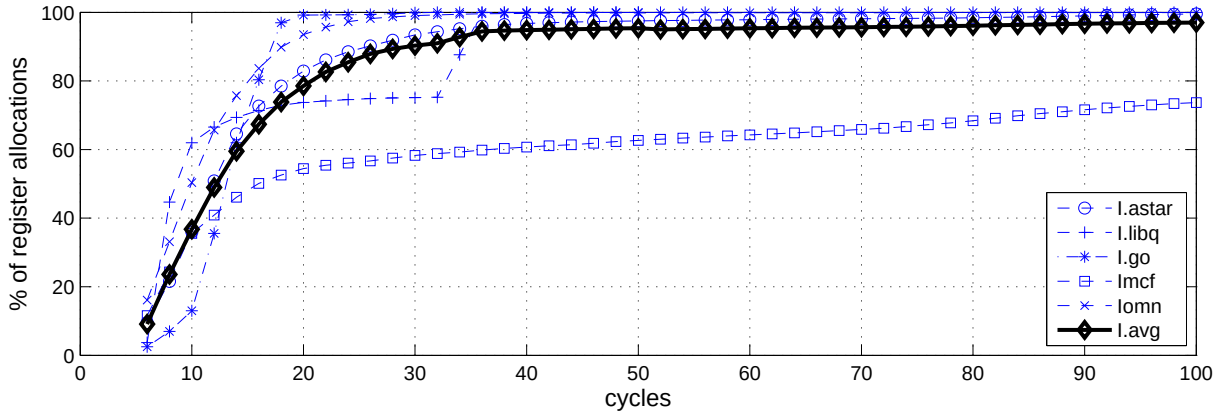
(b) SPEC2006 Integer

Figure 4.3: Register file occupancy. CDF showing fraction of time registers are occupied.

reducing read-port pressure, but *increasing* write energy and register capacity pressure. In (d)—the scheme used in this dissertation—register file banks are duplicated with their full compliment of read- and write-ports. This allows any functional unit to access any register in the register file, while avoid stalls due to contention and bank conflicts. However, this is at a cost of area and energy, as these ports add capacitance to the word and bit lines for each register. The benefit of this approach is that there is no performance penalty or bottleneck induced as in (b), no overhead in capacity due to duplication as in (c), and the dynamic energy costs are still much lower than in (a).



(a) SPEC2006 Floating Point



(b) SPEC2006 Integer

Figure 4.4: Allocate-write distance. CDF showing the distance in cycles between between preg allocation (cycle 0) and preg use at writeback.

4.1 Register File Utilization

Contemporary processor have very large register files in order to support a large instruction window and a large number of threads. However, application phases do not always exhibit high amounts of instruction level parallelism and often leave a significant portion of the register file dormant. Figure 4.3 shows a cumulative distribution function (CDF) of register file occupancy across SPEC2006 benchmarks for a register file with 160 64-bit registers. The CDF approaches 100% of run-time as the register file approaches 100% occupancy. The floating point workloads (top) typically have curves biased towards the bottom right, indicating that the spend more run-time with a higher fraction or the register file occupied. This is in contrast to the integer workloads (bottom), which are biased

more towards the top left, indicating that they spend more time with fewer registers occupied.

On average, only *68%* of the register file (108 registers) is in use for integer workloads and *84%* of the register file (135 registers) for floating-points workloads. Integer workloads have lower ILP due to less predictable branch-behaviour—register pressure is reduced as speculative in-flight instructions are squashed if they are executing down the wrong path. Floating point workloads have more predictable branches and larger amounts of ILP; with more instructions in flight, they require more registers.

In addition to this utilization slack, even registers that are "occupied" do not always contain valid state. The register file is a resource that is allocated early in the instruction pipeline—at register-rename—and first used late in the pipeline—at register writeback. Figure 4.4 shows the the number of cycles between register allocation at the register-rename stage and register-use during the writeback stage. This dissertation calls this figure of merit the "allocate-write" distance.

A minimum of six-cycles are needed between allocate and writeback due to the number of pipeline stages between rename and writeback. After an instruction is renamed, it is dispatched to the issue-queue—which may be full—where it waits for its source operands to be ready. If the source operands depend on a long-latency operation—such as a load that misses the cache, or a long floating-point divide—then the instruction will remain in the issue-queue and the allocated register will be allocated, but idle. When all source operands are ready, there is a minimum of three cycles between issue and writeback, called the "issue-write" distance. The interaction between the latency inherent due to the pipeline and the dynamic instruction stream yields two slacks that can be exploited for energy reduction: slack in the *size* of the register-file, and slack in *timing* when a register needs to be available after allocation.

4.2 Power Gating Opportunity

When register files are banked, each bank can be a candidate for power-gating when the bank is empty. Figure 4.5 shows the opportunity for power gating at different granularities—from single registers to banks of 4-,8-,and 16-registers—by recording the amount of time that the register-file bank is empty. The maximum opportunity arises when banking only a single register. However,

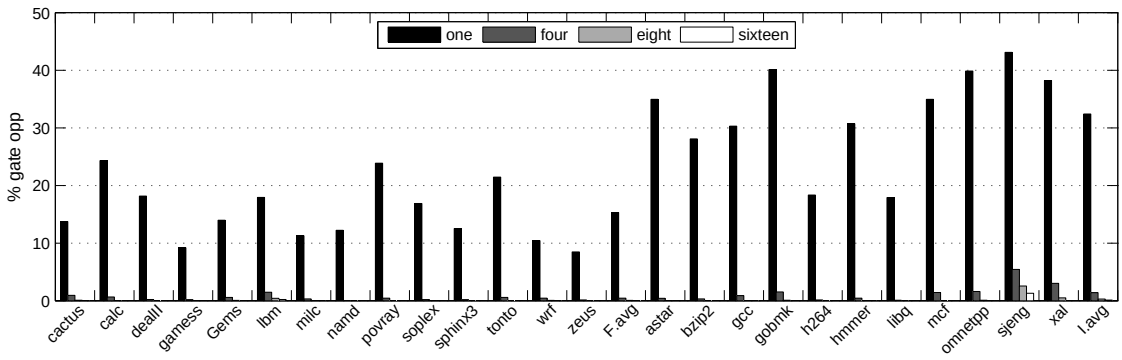


Figure 4.5: Gating opportunity. Percent of registers that can be disabled in SPEC2006 benchmarks for different gating granularities: individual registers, and banks of 4-,8-, and 16-registers

Table 4.1: RF leakage components for 160-entry, 6 read-port, 3 write-port, 64-bit register file

Component	I_{leak}	**%**
Precharge	174 uA	31.67%
SRAM	152 uA	27.67%
Buffers	142 uA	25.95%
Word Line Drivers	76 uA	13.90%
Decoders	2.57 uA	0.47%
Sense Amplifier	1.90 uA	0.35%

single register banks are not practical, as only the SRAM bit cell can be power-gated, rather than the entire bank including the periphery circuit.

The opportunity for power gating arises from how registers are aligned within the register file and where new registers are allocated. There is negligible opportunity to power-gate coarse grained banks of registers when allocation uses a conventional free list. Even though there is a significant opportunity for power-gating single registers, these registers are not aligned with coarse-grained register-file banks are not exploitable for power savings. Power gating individual registers within a register-file is not as lucrative. Table 4.1 shows the leakage components of a monolithic SRAM. It is the periphery circuitry (sense amps, decoders, muxes, drivers) which consume the most leakage energy. This is exacerbated by banked designs for clock gating, where there is a duplication of peripheral circuitry.

Physical registers are allocated to instructions during the rename stage of the out-of-order pipeline. In conventional processors, dispatching instructions are allocated a destination register

from the head of the free-list When an instruction commits its value to the architected state, the overwritten register is freed, and enqueued to to the tail of the free-list. Even if the processor can dynamically re-size the free list and take some registers out of circulation, the remaining registers are likely to be distributed evenly across the register file because the "un-disciplined" register overwriting continuously shuffles the contents of the free list. As the program executes, this FIFO allocation will "shuffle" registers across the register file. When a register is un-allocated, it will be the last register to be allocated to a new instruction, even if the register location makes the most sense for power-gating. As such, the FIFO approach is a power-gating "unaware" design.

This dissertation investigates methods to improve the gating opportunity for coarse grained banks that are designed with power-gating in mind. Allocation can be improved through register "packing", an operation that physically copies data from one register to another register that is more optimal for power-gating. This introduces more dynamic instructions into the instruction stream with some potential overheads. Another approach investigated by this dissertation is to make better choices for where to place registers in the register file when the register is allocated. This dissertation leverages reference counting register management and investigates algorithms which naturally pack registers, allowing power-gating to be applied opportunistically. These algorithms read the state of the register file from the reference count bitvector and make better decisions about how to use the register file.

4.3 Banked Register File Implementation

This dissertation uses the FabMem SRAM memory generator tool from the Fabscalar toolkit [12] to model register files and other SRAMs. This tool generates HSPICE netlists in a 45nm technology [90] which can be used for detailed timing and power simulations and generates estimates for area.

Table 4.2 shows the characteristics of register files built from banks of four-, eight-, and sixteen-registers. The table also describes a single-banked monolithic register file with 160 registers. Smaller banks have shorter read latency and lower read and write energy costs. This is due to the reduction in word- and bit-line capacitance. Banked register files have shorter wires requiring smaller drivers. The total area is significantly higher when using smaller banks due to the replication of periphery

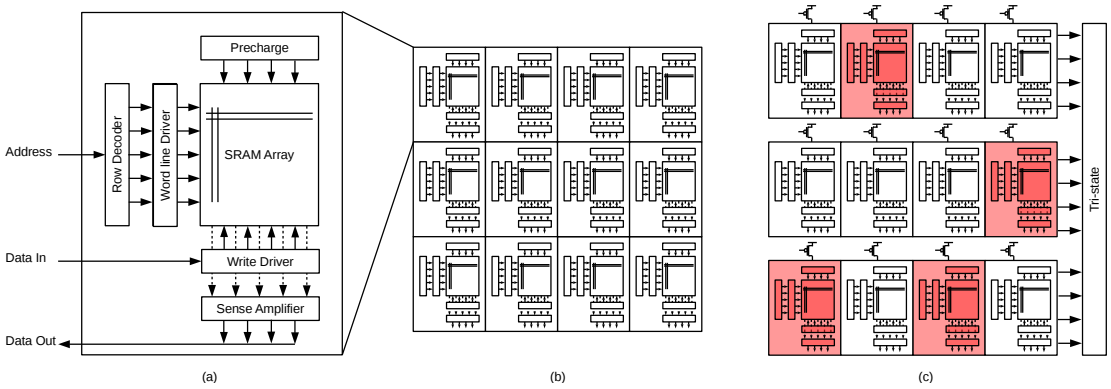


Figure 4.6: Banked register file block diagram. (left) Register file consists of SRAM bit cells, address decoders, sense-amps, and drivers. (middle) Banked register file contains multiple copies of each bank, with redundancy in address decoder, driver, and sense-amp circuitry between banks. (right) power-gating is supported by adding power-gate PMOS transistors and tri-state drivers to isolate banks. Disabled banks are shaded

Table 4.2: Register file costs. Area and energy costs for register files comprised of banks of 4-, 8-, 16-registers and a monolithic 160-entry register file, each with 6-read and 3-write ports. I^*_{max} measures current when 6-reads and 3-writes are active at a time over 0.625 ns (2-cycles at 3.2 GHz). 160-entry RF is measured during 1-cycle at min latency (1.56 ns).

Bank size	*Bank area*	*No. banks*	*Total area*	E_{read}	E_{write}	t_{read}	I^*_{max}
4	3844.5 μm^2	40	152780 μm^2	1.61 pJ	2.38 pJ	0.39 ns	28.0 mA
8	5484.5 μm^2	20	109690 μm^2	1.83 pJ	2.56 pJ	0.45 ns	31.1 mA
16	8774.9 μm^2	10	87449 μm^2	2.31 pJ	3.08 pJ	0.54 ns	38.5 mA
160	70113.9 μm^2	1	70114 μm^2	10.81 pJ	15.1 pJ	1.56 ns	70.6 mA

circuits—sense-amps, drivers, decoders, muxes, and ports. A 160-entry register file built out of banks of four has an area overhead in excess of 2.1× a monolithic register file, while banks of sixteen registers has an overhead of only 1.2×. As monolithic register files do not support clock-gating, for the rest of this dissertation we only consider banked registers files in our experiments. Techniques have been proposed to reduce the banking overhead by reducing the port density [60, 75], however these approaches introduce scheduling overheads as each bank cannot support the full execute-bandwidth of the processor. In this dissertation, each register file bank supports the full port-width of the monolithic register file so as to prevent the banks themselves from becoming a bottleneck. The techniques described in the following sections will also work on port-reduced register files.

4.4 Reference Count Power Gating

Reference counting supports a straightforward interface for power-gating register file banks. The key requirement for power gating register file banks is that the bank must be empty. A simple OR gate (or NOR gate) circuit reads the bit-vector for a particular bank and determines if the bank has any registers allocated to it. When the bank has no registers allocated, the vector will be all zeroes and the bank is a candidate for power gating. A NOR gate will generate a '1' from this vector input, which can be used to drive the PMOS power gate transistor.

A control signal will also be attached to the 'OR' logic, as even though the bank may be empty, a power-gate controller may want to add hysteresis to keep the bank enabled and prevent toggling of banks. Recall that you do not want to disable a bank only to power it on before it has reached the 'breakeven' point. Figure 4.7 illustrates an example implementation of reference count power gating. In this figure, the bank on the right is empty and disabled. The bank on the left has registers allocated to it and will not be power gated.

One important issue is the relationship of gating banks to allocation banks. In a four-wide superscalar processor, the register file is divided into four allocation sets. To maximize the number of empty banks, the allocation sets are interleaved relative to the banks—bank 0 covers registers 0–7, allocation set 0 covers registers 0, 4, 8, etc. Up to four instructions per cycle may be allocated registers from their own free-list (or reference count). If the allocation were not partitioned then the reference count vector or freelist would require more ports. Interleaving allows each 'way' in the processor to be allocated from the same register bank.

4.4.1 Power Gating Implementation

Power gating is implemented with few modifications to the register file banks. First, the PMOS gate is inserted between the Vdd node and the SRAM circuit. A set of inverters drives the power gate enable signal to this PMOS. These inverter drivers control the input to the PMOS gates, and therefore exist out of the power-gate voltage domain. They cannot be power gated and are connected to VDD directly. These inverters contribute to the cost of power gating, as it is their switching action

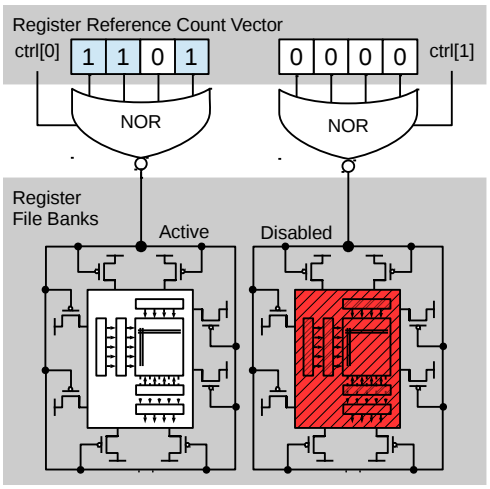


Figure 4.7: Reference count register bank power-gating. Per-bank NOR gate reads vector. If vector is empty PMOS gate signal is driven high, disabling the bank.

Table 4.3: Power-gating overhead. PMOS gate width is calculated according to equation 4.1. We compare the energy saved due to the reduction in leakage current when power-gated vs clock-gated. Breakeven time measures the number of power-gated cycles required in order to recover the cost of toggling the bank.

Regs	$I_{clk-gate}$	$I_{vdd-gate}$	W_{pmos}	A_{oh}	E_{switch}	*Breakeven time*
$4 \times 64b$	39.6 μA	87 nA	8.5 μm	0.33%	0.59 pJ	26 cycles
$8 \times 64b$	59.1 μA	199 nA	11.3 μm	0.35%	0.85 pJ	25 cycles
$16 \times 64b$	97.4 μA	206 nA	18.6 μm	0.37%	1.36 pJ	24 cycles

in addition to that of the PMOS gate which constitutes the switching overhead.

The size of the PMOS power-gate transistor is important to the performance of both the power-gating logic *and* the active logic. The additional PMOS transistor adds another node between $V_{dd}$and ground, increasing the effective 'stack-height' of the transistors. A larger 'stack' will have lower leakage current when the circuit is disabled. A stack of disabled transistors will have significantly reduced leakage current compared to a single transistor due to a reverse bias between the gate and source [11].

However, stacking transistors can limit the active current through the circuit; the extra transistor must be sized carefully in order to limit any degradation of logic switching speed. There are two requirements for the PMOS power gate transistor—first, the latency should not exceed the original cycle time requirements; second, the circuit must be able to restore the V_{dd}signal before the register

bank is accessed. We calculate the size of the PMOS gating transistor using Equation 4.1 [80]. This sets the gate width according to the acceptable additional delay caused by the increased stack-depth (power-gate delay, PGD) and the maximum current through the active circuit (I_{on}). Additional parameters are technology dependent and include supply voltage (V_{dd}), PMOS gate resistance (R_m), and transistor threshold voltage (V_t).

$$W_{\mathrm{P}MOS} = \frac{1}{1 - \sqrt[1-\alpha]{PGD}} \left(\frac{R_{\mathrm{m}}}{V_{\mathrm{dd}} - V_{\mathrm{t}}} \right) \times I_{\mathrm{on}} \tag{4.1}$$

The PMOS gate width is calculated assuming a 3% increase in access latency can be tolerated. This is still within the two-cycle register-file access latency for the baseline processor. The maximum current is measured assuming all ports in a single bank (six read-ports and three write-ports) are active. The PMOS gate is adjusted after simulations to support a two-cycle Vdd recovery, where the Vdd signal recovers from a power-gated state to a voltage exceeding 0.9V (90% of Vdd) in 2-cycles and 0.95V after 3-cycles. This yields the PMOS widths listed in Table 4.3. The PMOS width also defines the size of the inverter drivers. A set of inverters drive the 'enable' signal from the reference count vector to the PMOS gate. This driver is sized from the PMOS gate for a logical effort of four and is included in the switching costs and area overhead columns in the table.

4.4.2 Breakeven

In order for power-gating to be profitable, the circuit must be power-gated long enough to recoup the cost to disable and re-enable the circuit, *i.e.* the switching energy overhead. Figure 4.8a—c shows the breakeven curves from HSPICE simulations of register file banks containing four-, eight-, and sixteen-registers. Each bank is disabled in cycle 1 and re-enabled on cycle 40. The breakeven point on the x-axis is highlighted for each bank size.

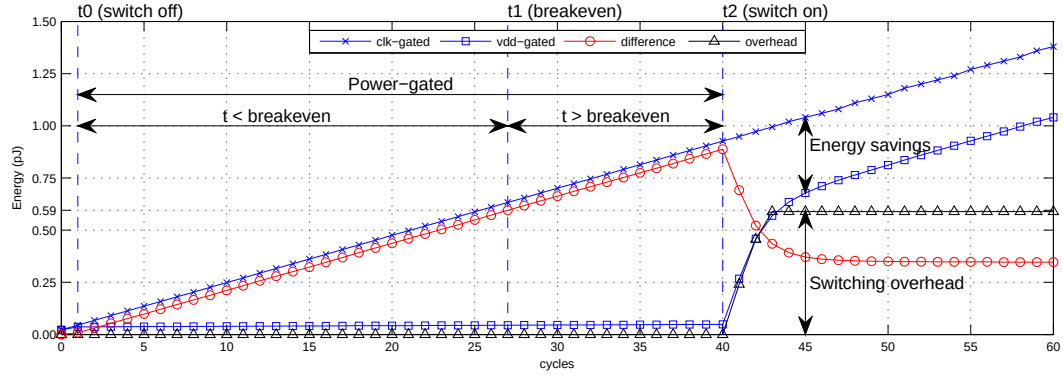
Each plot shows four lines: the clock-gated energy accumulation, the Vdd-gated energy accumulation, the difference between clock- and vdd-gated energy, and the switching overhead. Clock gating the bank is the alternative low-power energy mode for the bank, and is our basis for comparison. When clock gated, all inputs to the bank are gated, and the only dissipated power is static leakage

power. We plot the accumulated energy per cycle for clock-gating as our baseline. When the bank is power-gated between cycles 1 to 40, a significantly lower amount of leakage current is dissipated due to the stacking-effect described previously. Table 4.3 lists the power-gated leakage current, which is approximately three orders of magnitude lower than the corresponding clock-gated current for each bank—nA vs μA. Breakeven time is measured as the time in cycles that the bank must be disabled in order to recoup the energy cost consumed by switching-off and switching-on the bank. When the 'difference' line is equal to the switching overhead, the bank has recovered its switching cost. This switching overhead is identified on each plot on the y-axis.

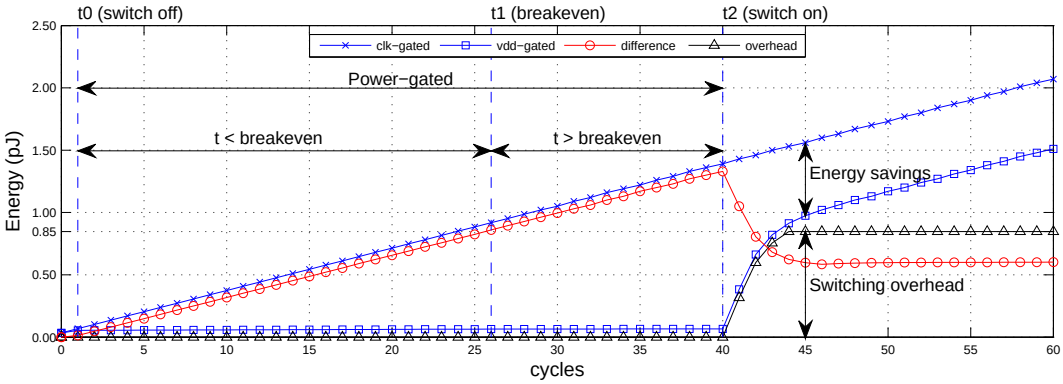
Figure 4.8a—c shows the breakeven time for each bank is in a narrow range: from 26-cycles for banks of four registers, to 24-cycles for banks of sixteen registers. This is expected as the PMOS width scales with the current through the circuit and the drivers are sized proportionally to the PMOS driver. Likewise, the measured area overhead shown in Table 4.3 is small and proportional to the area of the bank, ranging from 0.33% in banks of four to 0.37% in banks of sixteen registers. Low overhead and short breakeven time are essential for efficient power-gating as there are more opportunities for energy reduction if the breakeven time is low.

4.5 Register Allocation Algorithms

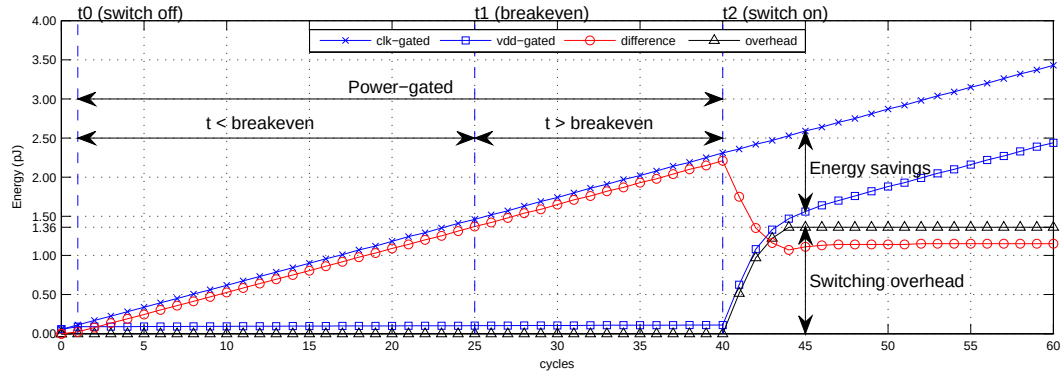
This section describes introduces novel algorithms for register allocation that will take advantage of the register file occupancy slack, to better support a dynamic power-gated register file that will save leakage energy. These allocation and bank gating algorithms couple microarchitecture information with circuit techniques, and focus on *where* to place registers and *when* to toggle register file banks in order to maximize energy reduction. This dissertation evaluates these allocation algorithms with respect to power-gating, but these approaches are also applicable to drowsy and retention based approaches to register file power reduction [28], where it is also important to know which registers are in use. Figure 4.9 illustrates each algorithm at a high level. In cases (a)—(c), a reference count bit vector is read by an encoder which selects the next available register. Each algorithm is evaluated against a conventional free-list approach shown in (d). The goal of these allocation algorithms is to maximize both the number of empty banks and the length of time that these banks are empty.



(a) Bank of four registers



(b) Bank of eight registers



(c) Bank of sixteen registers

Figure 4.8: Power-gate breakeven. Energy cost to power-gate a bank of registers compared with energy consumed by a clock-gated bank.

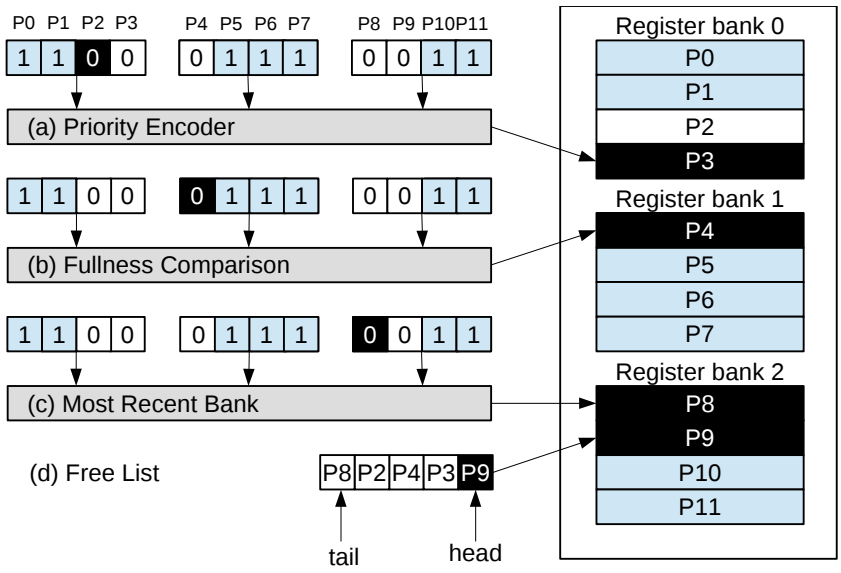


Figure 4.9: Allocation algorithms. Each algorithm examines the set of available registers to select the next register for allocation. (a) Free-list: selects the reg at the head of the FIFO queue. (b) Prio: select first free reg from a bitvector representation of the RF ('0'=free, '1'=allocated). (c) Full: select first free reg from fullest RF bank. (d) MRA: select first free reg from most-recently-selected bank

4.5.1 Priority Encoded

Figure 4.9a shows a priority encoding scheme where registers are allocated to the first free register in the register file, identified by an empty bit in the reference count bitvector. In a four-way superscalar processor, the register file is partitioned among each 'way' in the processor. This requires a set of four priority encoders. In a processor with 160-registers, each 'way' can allocate from a 40-register partition. The priority encoder reads the state of the 40-bit vector and generates a 6-bit binary encoded register value.

In this and the following schemes, registers are not allocated in the order that they are freed as is done in the FIFO scheme. Registers in a priority encoded allocation will pack towards the lower-order end of the register file, allowing the upper order registers to be disabled after they are freed. The lower order entries have a higher priority for allocation than the upper order registers.

4.5.2 Fullest Bank

The fullest-bank scheme shown in Figure 4.9b modifies the priority-encoded allocation by selecting the first available register from the *fullest* bank, rather than the first available register from *all* banks. An implementation is shown in Figure 4.10 where the reference count bitvector is partitioned according to register-file bank width. Each vector is examined to determine both the number of available registers (its fullness), and the next available register in that bank. Each n-bit vector requires an n-bit to $\log_2$ n-bit priority encoder and n-bit zero-counter. An n-bit comparator compares the fullness of adjacent banks, steering the fuller bank to the output. Components routing the fullest bank (bank 3) are highlighted in grey.

An example processor may have a 160-entry register file and twenty banks of eight-registers. Each 'way' in the four-way superscalar processor would require five 8-bit priority encoders, zero-counters, and a mux-tree to propagate the selected bank and register/ This mux-comparator tree is an additional cost over the priority scheme; however the priority encoders are much smaller, requiring 8-bits compared to 40-bits for the priority case.

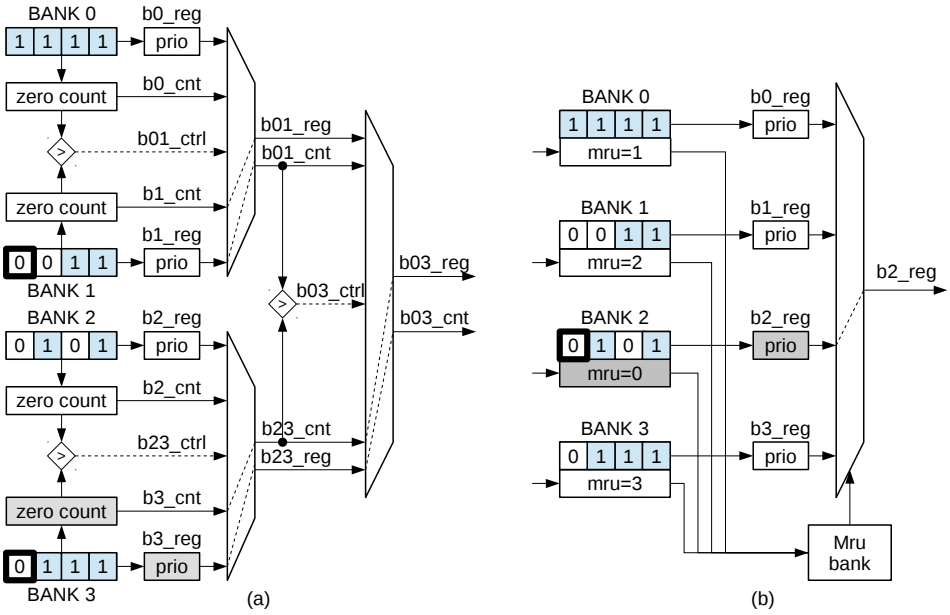


Figure 4.10: (a) Fullest bank allocation. (b) Most-recent bank allocation.

4.5.3 Most-Recently Allocated (MRA)

The most-recently allocated bank shown in Figure 4.9c keeps a chronological history of each allocation similar to the LRU stack in a typical LRU cache controller. The goal of this allocation scheme is to group instructions of the same age together in the same bank. It is expected that these instructions will have a similar lifetime, affording greater opportunities to power-gate than traditional free-list allocation. If the MRA bank is full, the next most-recent bank with space is selected. When a bank is selected, its MRU register is set to 0, indicating it is the most-recent selection. The MRU register in every other bank whose MRU position was lower than the selected bank is incremented. The priority encoded value of the banks register reference count is selected via a $\frac{num.\ Pregs}{num.\ Ways}$ wide mux, illustrated in Figure 4.10b.

4.5.4 Implementation

Each scheme described above is implemented in a Synopsys RTL flow in Verilog, and synthesized using NCSUs 45nm design kit. Testbenches simulate average activity rates recorded from an $x86$ simulator. These testbenches collect average and maximum activity used for power simulation in

Table 4.4: Register allocation algorithm costs

Design	*Area*	P_{dy}	P_{st}	P_{av}
Register File 6r3w $160 \times 64b$	108733 μm^2	22.70 mW	2.49 mW	9.97 mW
Freelist $4 \times 27 \times 32b$	2279 μm^2	2.08 mW	0.06 mW	0.82 mW
Priority 4r4w $1 \times 160b$	7622 μm^2	1.24 mW	0.03 mW	0.72 mW
Fullest (8-reg banks) 4r4w	11618μm^2	1.95 mW	0.06 mW	0.81 mW
Most-recent (8-reg banks) 4r4w	11618μm^2	1.95 mW	0.06 mW	0.81 mW

prime-time PX. The logic is synthesized at a design target frequency of 1GHz and power is scaled to match the design frequency of 3.2GHz for the performance simulator.

The synthesized logic consumes more area than the freelist SRAMs for three reasons: first, complex logic gates are less dense than the SRAM cells; second, the 1GHz target frequency is aggressive in this technology kit, requiring large library cells; and finally, the area consists of mostly complex decoder and encoder circuits which is much larger than the S-R latch bitvectors. Despite this, the average and static power however are on the same order as the freelist SRAM. Only four-bits in the vector are toggled at a time, rather than four six-bit values in the freelist. As expected, the fullest bank allocation scheme consumes the most power out of the algorithms under study, as it requires both zero-counters and encoders, while the priority scheme only requires encoders, and the most-recent scheme requires encoders and a register per bank.

4.6 Register Gating Algorithms

The goal of a register bank gating algorithm is to maximize both the number of banks that are disabled and the number of cycles that a bank is disabled. Toggling is to be avoided, as it will cause banks to be enabled prior to reaching the break-even point, thus costing more energy than it saves.

4.6.1 Immediate

The first algorithm disables the bank *as soon as possible*. Once the bank is empty, the gating signal will be asserted and the bank disabled. This has maximum opportunity for gating banks, but also maximum potential for thrashing or toggling register banks. The allocation and gating algorithms are decoupled, and disabled banks are still candidates for allocation which allows for banks to be

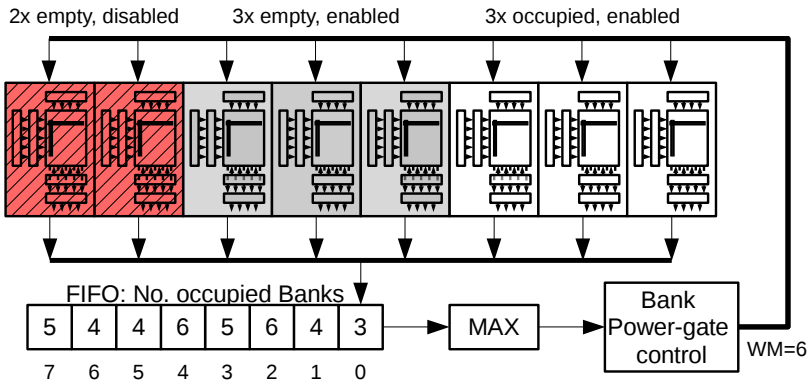


Figure 4.11: Watermarked Gating. Controller keeps a count of the number of active banks during 8-cycle windows, storing in an 8-entry FIFO. The maximum number of active banks during previous 8 windows is used as the low watermark. Empty banks in excess of the watermark disabled. 5-banks are empty and 3 are occupied. The watermark is 6, so only 2 banks are disabled.

enabled immediately after being disabled. However, this algorithm naturally aligns well with the *fullest bank* allocation scheme, as an empty bank is only selected for allocation and re-enabled when every other bank is full. This means that empty banks both powered off *and* are the lowest priority for allocation.

4.6.2 Watermarked

This algorithm, shown in Figure 4.11, keeps track of the number of active banks over the previous eight windows, where each window is eight-cycles long The high watermark out of eight counters is recorded, and all enabled but empty banks in excess of the watermark are disabled. This conservatively tracks the register usage, with a goal to keep register banks disabled in excess of the breakeven time. The cost is in potentially missing opportunities to gate more banks and save leakage power. If insufficient banks are enabled when an instruction needs a register allocated, the empty bank is enabled immediately at rename when the register is allocated, setting a bit in that banks bitvector.

4.6.3 ROB-proportional

This algorithm enables banks in proportion to re-order buffer (ROB) occupancy. The ROB contains an entry for every in-flight instruction. As ROB utilization increases, more instructions are in flight

and accessing the register file. A less occupied ROB will require fewer banks to be enabled than a more occupied ROB. This scheme scales the register-file with ILP. As ILP increases, more register banks are enabled because ROB utilization will increase. When instructions (and their ROB entries) are squashed or committed, utilization decreases and empty banks are disabled. For example, if the ROB is only 50% full, up to half of the banks are disabled if they are empty.

There is one exception to the proportionality, which covers the case when the processor stalls due to lack of ROB occupancy. If there are fewer entries in the ROB than the current dispatch width, then the processor is about to stall. No forward progress can be made if the ROB is full. This can occur when a load instruction misses the last-level cache or when a long-latency instruction—such as a square-root or divide—is at the head of the ROB but still executing. Once the re-order buffer has fewer than entries then the dispatch width, all currently empty banks are *disabled*. While the ROB is full, a stall condition is occurring. When the ROB is unblocked and instructions retire, registers are freed by other committing instructions. This frees up space in the currently enabled banks, allowing the disabled banks to remain empty.

4.7 Evaluation

This dissertation uses a cycle-accurate x86 simulator coupled with HSPICE circuit simulations to evaluate the reference counting approaches to register management and Vdd gating. The x86 simulator executes 64-bit user-level code. The baseline configuration is a conventional out-of-order processor design with a re-order buffer, similar to Intel's Core i7 (Nehalem) processor. The salient features of the processor for this evaluation are the superscalar width, instruction window size, and register file size. The processor models a four-wide superscalar width, able to issue four instructions per cycle from the 36-entry issue queue. Up to four instructions per cycle can be retired from the 128-entry re-order buffer. The processor has 64KB instruction and data caches, a 128KB L2 cache, and a 2MB L3 cache. The register file has 160 physical registers—96 rename and 64 architected—with a two-cycle register read latency. The power-gating techniques presented do not cause the register read latency to exceed this two-cycle limit, so performance is identical between all configurations. Chapter 6 and Table 6.1 describes the baseline Nehalem processor in more detail.

Table 4.5: Benchmarks

Benchmark	Input	IPC	Avg. Regs	BP Acc	L2 MLP	L2 MPKI	L3 MLP	L3 MPKI
cactusADM	test	1.07	129.2	94.1%	4.10	4.73	4.42	2.75
calculix	train	1.53	109.5	96.0%	1.94	2.18	3.75	0.52
dealII	test	1.62	119.7	97.3%	1.83	2.66	3.52	0.30
gamess	test	1.68	134.7	96.3%	1.42	1.63	2.47	0.10
GemsFDTD	test	1.08	128.2	97.3%	2.42	10.09	4.04	3.00
lbm	test	0.59	127.3	98.8%	4.99	38.81	4.39	18.13
milc	test	0.50	138.2	98.4%	2.46	24.44	2.74	14.64
namd	train	1.55	127.5	96.4%	1.73	0.14	2.91	0.03
povray	train	1.38	109.1	94.0%	1.55	0.06	2.86	0.00
soplex	train	0.59	127.9	95.6%	2.21	24.27	2.91	8.23
sphinx3	train	1.27	131.3	96.8%	2.02	14.16	3.46	1.17
tonto	test	1.66	112.6	97.1%	1.43	0.20	2.14	0.03
wrf	test	1.44	134.6	98.9%	2.16	5.81	4.33	0.48
zeus	test	1.00	139.7	98.3%	1.98	6.83	3.59	2.69
astar	test	0.83	95.1	83.0%	2.01	5.29	2.20	0.05
bzip2	train	1.44	101.4	93.9%	2.31	3.28	2.52	0.08
gcc	train	1.21	99.1	95.7%	1.39	3.37	1.90	0.49
gobmk	train	0.98	86.6	85.8%	1.50	2.03	1.65	0.20
h264ref	test	1.90	112.5	97.7%	1.43	1.76	1.66	0.05
hmmer	test	1.36	97.4	92.4%	1.36	0.00	2.45	0.00
libquantum	train	1.97	116.3	98.7%	3.01	21.96	3.53	0.00
mcf	train	0.21	102.3	95.2%	2.53	112.37	2.35	17.62
omnetpp	test	1.22	84.6	90.8	1.26	0.17	1.48	0.12
sjeng	test	0.96	82.2	88.4%	2.96	7.71	7.67	1.40
xalancbmk	test	1.22	87.1	91.4%	1.62	2.62	2.99	0.14

4.7.1 Benchmarks

This evaluation uses a mix of 24 integer and floating-point workloads from the SPEC2006 benchmarks. Table 4.5 describes the benchmark characteristics for the floating point (top) and integer (bottom) workloads. These benchmarks were compiled with gcc-4.1 with -O3 optimization flags. The benchmarks in this evaluation were executed with a mixture of test and training inputs, depending on the execution length. The simulator periodically samples the benchmarks, for a detailed analysis of 2% of the entire workload. Each sampling period is 500 million instructions long with a 50 million instruction interval between periods. The period is divided into three regions: 10 million instructions of cache and predictor warm-up, 10 million instructions of detailed timing simulation, and 480 million instructions of functional simulation.

These workloads have different characteristics and exercise the register-management hardware in

different ways. Floating point workloads typically have higher branch prediction accuracy, from 94.1-98.8%. This accuracy and predictability means that ILP is typically higher, and fewer instructions are squashed by executing down the wrong path of a branch. This allows more instructions to enter the instruction window without being squashed, and causes higher register pressure. However, high branch prediction accuracy does not mean that IPC will be high, as there can still be significant latency due to memory operations. Workloads with frequent cache misses—*i.e.* higher L2 or L3 MPKI—will experience stalls as the ROB and issue queue fill up with instructions dependent on a load miss. Integer workloads have lower branch prediction accuracy and a higher fraction of squashed instructions, reducing pressure on the register file. In many case, integer workloads use fewer than 100 registers, while floating point workloads all use more than 100 registers on average.

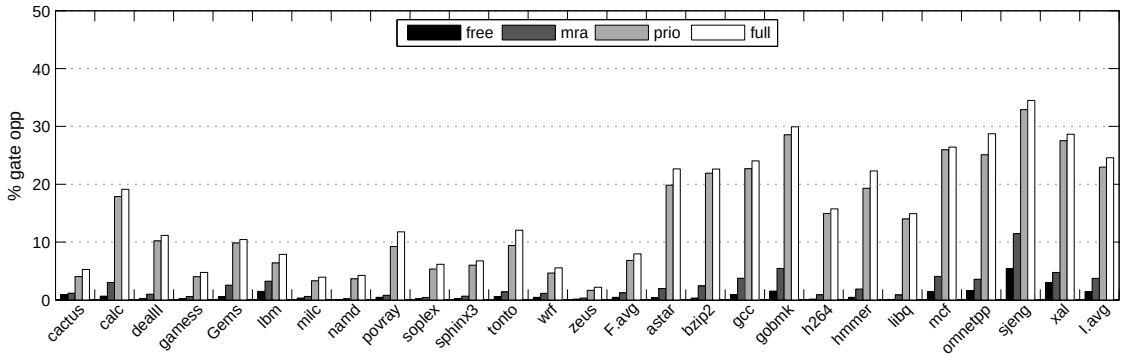
4.7.2 Allocation Algorithm Comparison

A good allocation algorithm will provide more opportunity for power-gating the register file by maximizing the number of free-banks. We compare the effectiveness of these algorithms by measuring the average percent of the register file that can be gated for banks of 4-, 8-, and 16-registers.

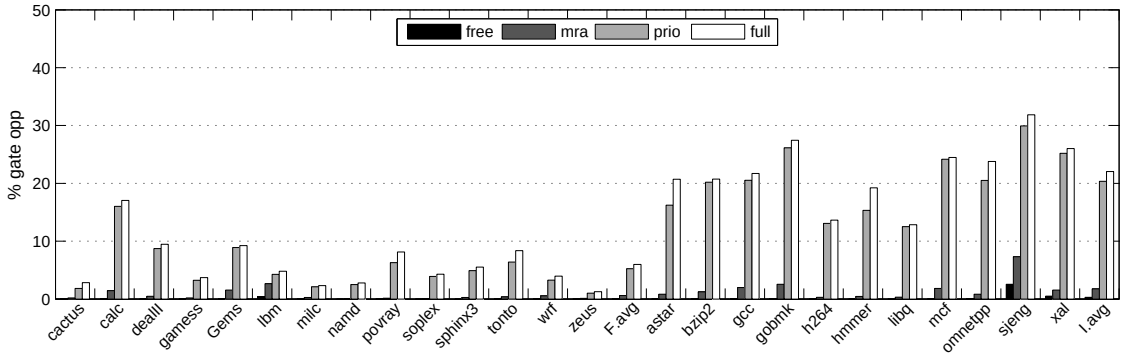
Figure 4.12 shows the percentage of the register file that is disabled for banks of four-, eight-, and sixteen-registers. In this comparison, register banks are disabled immediately when the bank is empty, so the figure represents the maximum opportunity that the register file could be gated. The combination of allocation, gating, and enabling algorithms will determine if the opportunity can be exploited for leakage reductions.

Conventional FIFO free list allocation is shown in the left columns in black. As expected, FIFO allocation does not perform well for coarse grained register file banks. Registers end up spread across the register file as register overwrite and allocation are decoupled operations. With no clustering to take advantage of, free list allocation leaves only 1.4% of the register file empty for SPECINT workloads and 0.45% of empty for SPECFP workloads for banks of four registers. Increasing the bank size to sixteen reduces the opportunity to disable register banks by over 10×, yielding a negligible fraction of 0.12% disabled for SPECINT and 0.02% disabled for SPECFP workloads.

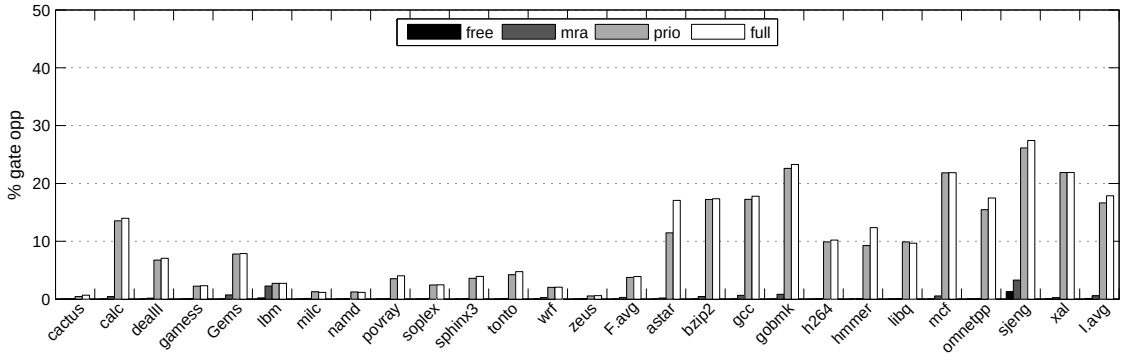
Figure 4.12 shows that most-recently allocated performs better than the free list, peaking at



(a) Bank of four registers



(b) Bank of eight registers



(c) Bank of sixteen registers

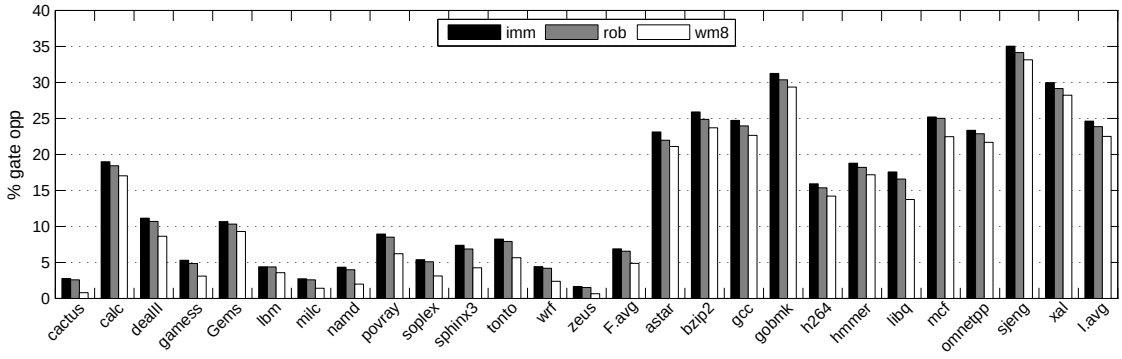
Figure 4.12: Allocation algorithm comparison, showing the average amount of the register file that is able to be power-gated

banks of four with 3.7% disabled for integer and 1.2% disabled for floating point workloads. This is still not a significant fraction of the register file compared to the average opportunity of 15% (FP) and 32% (INT) for single registers shown earlier in Figure 4.5. Registers are still scattered in this MRU scheme, especially when register pressure is high. This scattering occurs because registers are not freed in the order that they are allocated. In some cases, allocating to the same bank for 'younger' instructions will not improve gating performance. In other cases, it is not possible to group instructions of the same age together when the most-recent banks become full.

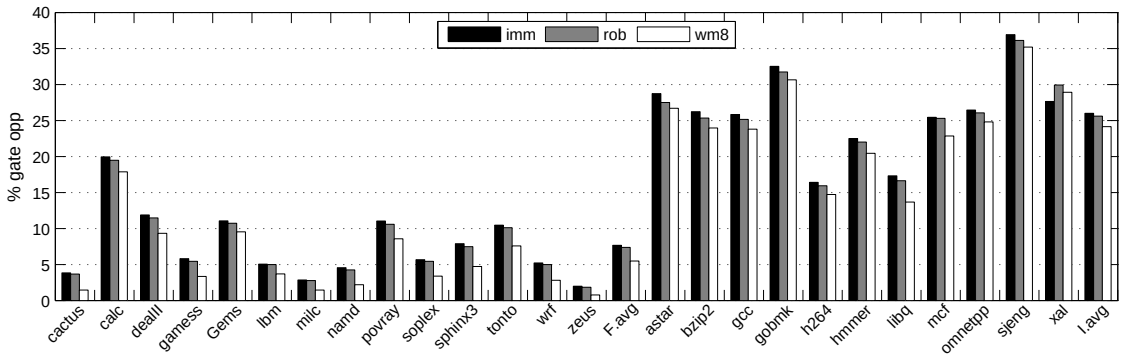
The priority-encoded scheme performs well, gating 6.8% and 22.9% of the register file for floating point and integer workloads when using banks of four registers. Registers are packed towards one end of the register file allowing higher order banks to be disabled. *Fullness* performs best overall, disabling 7.9% of the register file during floating point workloads and 24.5% during integer workloads, an average improvement of 16% and 7% over the *priority* scheme for floating point and integer workloads. *Fullest*-bank allocation improves over *priority* by eliminating cases where new register allocations re-enable empty banks because they contain the *first* empty register and cases where the other bank occupants are freed soon after the new register is allocated. The performance for all algorithms attenuates as bank-size increases, but most-recent is particularly affected, with less than 1% of the register file disabled for sixteen-entry register file banks. However, *fullest* and *priority* bank allocation still capture some opportunity to disable register banks. For workloads such as `astar`, the disparity between fullest and priority increases, indicating that fullest allocation is making better decisions.

4.7.3 Bank-Gating Algorithm Comparison

Algorithms that control *when* a register bank is disabled will affect the leakage energy of the register file bank and the fraction of bank toggles which are profitable and exceed the breakeven time. An aggressive algorithm my disable register file banks more frequently, but if those banks are only disabled for a short period of time, then the scheme could exhibit a net loss of energy compared to a conventional clock-gated approach.



(a) Priority allocation

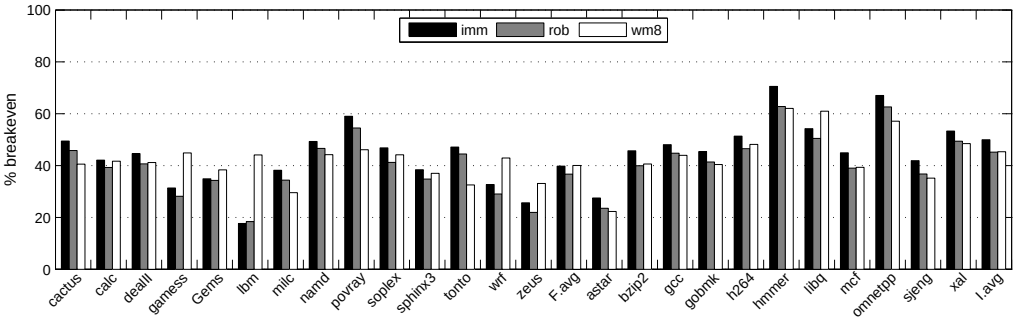


(b) Fullest allocation

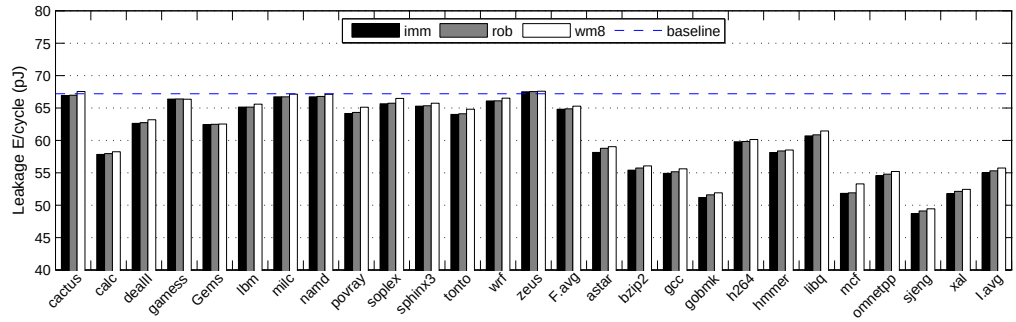
Figure 4.13: Gating algorithm comparison for banks of eight-registers

Fraction Gated

The first metric affected by the gating algorithm is the average amount of the register file that is disabled. A conservative algorithm will disable fewer banks, keeping some banks enabled to be used by new instructions. Aggressive algorithms will try to disable the register banks as soon as they are empty. Figure 4.13a shows the fraction of the register file that is disabled when sweeping the gating algorithm while keeping the allocation algorithm fixed at priority encoded allocation. Figure 4.13b shows the fraction of the register file that is disabled when sweeping the gating algorithm while keeping the allocation algorithm fixed at fullest allocation. We do not consider the free-list or most-recent allocations because the power-gating opportunity is significantly constrained by register scattering.



(a) Toggles exceeding breakeven time



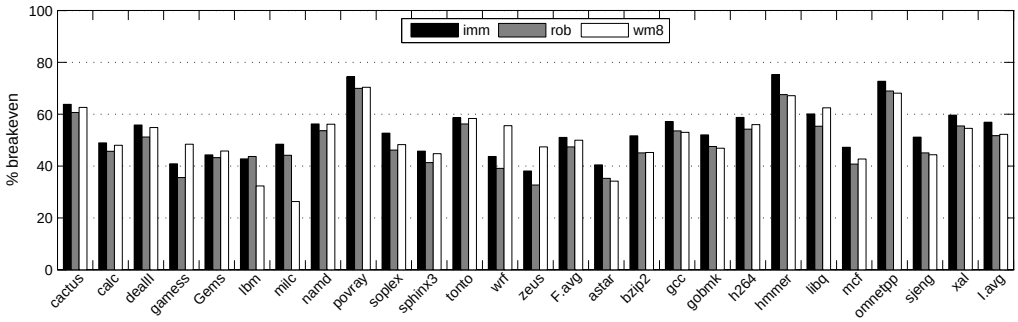
(b) Leakage energy per cycle

Figure 4.14: Gating algorithm comparison for banks of eight-registers allocated with priority algorithm

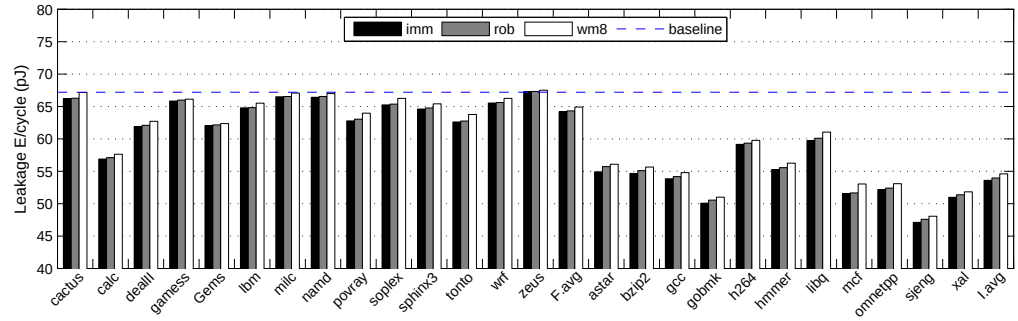
In terms of fraction gated, the *immediate* gating algorithm performs best, as banks are disabled once their reference count is empty. The disabled bank has the lowest priority to be re-enabled because any active bank will have higher occupancy and be preferred. *Watermark-8* has the lowest amount disabled as its watermark approach is slower to track changes in program behavior. The *ROB-proportional* approach performs similarly well with as the *immediate* algorithm as ROB occupancy can function as a proxy for register pressure. These trends are the same for both Figure 4.13a and b.

Breakeven and Leakage Energy

Figures 4.14 and 4.15 apply our energy model from HSPICE simulations to the simulator performance model, associating a per-cycle energy cost for each cycle of power-gating. With this information, we can measure both the fraction of toggles that break-even and the per-cycle leakage energy compared to clock-gating the register file bank. These figures show the breakeven time and leakage energy per



(a) Toggles exceeding breakeven time



(b) Leakage energy per cycle

Figure 4.15: Gating algorithm comparison for banks of eight-registers allocated with fullest-bank algorithm

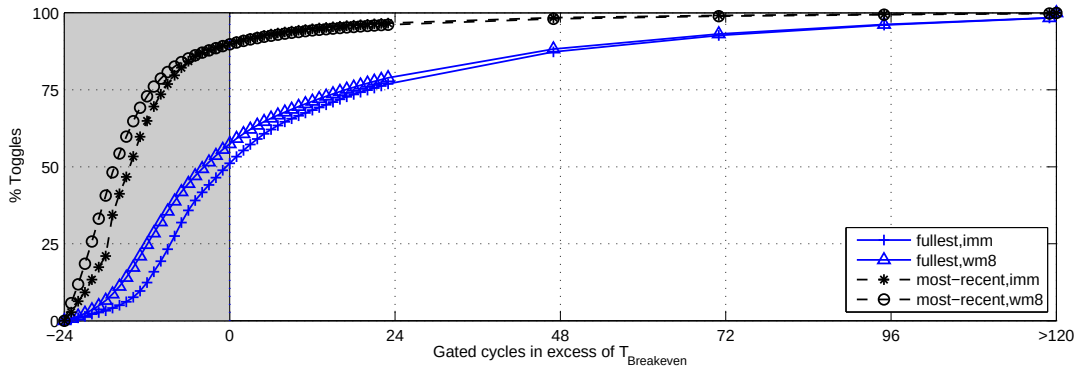


Figure 4.16: % Breakeven CDF. Cycles spent disabled in excess of breakeven time for sjeng benchmark and two different allocation and gating configurations

cycle for each gating algorithm applied with the priority and fullest-bank allocation algorithms. The leakage energy per cycle is measured across the *entire* workload. The default clock-gating leakage current is applied to cycles when a bank is enabled, as there is still leakage current associated with normal operation of the bank. The delta between clock-gating and power-gating is only applicable during the fraction of the workload when the bank is power-gated.

Figures 4.14a and 4.15a show how gating algorithms and allocation algorithm affect bank breakeven rates for register banks allocated using priority-encoded and fullest-bank algorithms. Note that you cannot infer the reduction in leakage-energy from these figures, just the efficacy of the gating decision. A bank may breakeven and be disabled for hundreds of cycles, saving significant amount of leakage energy vs clock-gating, while a bank may miss breaking even by only one cycle without a significant energy cost. In Figure 4.14a, immediate gating is typically the best performer, though watermarked gating works particularly well for `lbm` and `gamess`. In these cases, the watermark algorithms guardband protects against a register bank toggling on and off too frequently. The immediate algorithm does not incorporate hysteresis, but it does take advantage of extra cycles that a bank can be disabled. In most cases, the watermark guardband is too conservative.

We see the effect of each gating algorithm on leakage energy in Figure 4.14b. This figure shows the leakage energy per cycle for the register file. Integer workloads (right) have lower energy per cycle have a larger fraction of the register file that is idle. The effect of gating algorithm is minimal because this energy is aggregated over the entire execution of the workload, including both when the bank is gated and when the bank is active. These figures show that while the watermark gating algorithm may have more toggles that exceed the breakeven time for some workloads, gating immediately allows a higher fraction of the register file to be power-gated and ultimately has the lowest energy costs.

Figure 4.16 illustrates the interplay of allocation and gating algorithm and the effect on the register file banks. There is a fixed opportunity to power gate a register bank resulting from the register occupancy at any given point in the workload. The `sjeng` workload has a maximum opportunity of of approximately 35% depending on the allocation algorithm (seen in Figure 4.13). Figure 4.16

shows a CDF of the amount of time the register bank is powered down in excess of the breakeven time. The x-axis extends to -24 cycles because it takes 24-cycles for banks of eight registers to breakeven—the bank is disabled at $x = -24$ cycles. When the line crosses $x = 0$, the toggle has finally reached the breakeven time. The time to the right of $x = 0$ is an energy recovery region, yielding a net reduction in energy compared to clock-gating that bank. Time spent to the left of $x = 0$ is an energy sink region, where the cost to disable and power on the bank exceeds energy spent clock-gating.

For the 'most-recent' allocation scheme with watermarked gating (solid line), 80% of the bank power-gate toggles are for a duration shorter than the breakeven time. These leaves very little opportunity to reduce power. WM8 is the most conservative gating policy, leaving plenty of banks enabled for incoming instructions. This lowers the opportunity for gating, but those banks that are disabled *should* be disabled for a longer time than However, the fullest allocation scheme with immediate power-gating (dashed line), has fewer than 50% toggles which do not break even. The curve is shifted down and to the right, indicating that more time is spent in the energy-recovery region.

4.8 Enhancing Power-Gating

Two approaches for improving register file power gating are evaluated. The first approach attempts to resolve the power-gating inefficiency that arises due to decoupled allocation and commit. Overwritten registers are freed when the writing instruction commits its result. As registers are de-allocated, orphan registers can be left in a register bank, keeping a bank enabled.

4.8.1 Compaction Moves

Even with a robust register allocation algorithm, there will still be times when a single register is preventing a bank from being powered down. Registers are only freed after the logical destination is overwritten. A given register may not be overwritten for a long time an could be effectively 'stuck' in a high order register bank. This is especially true after pipeline flushes when in flight instructions that could overwrite the register are squashed. This also occurs during long-latency cache misses,

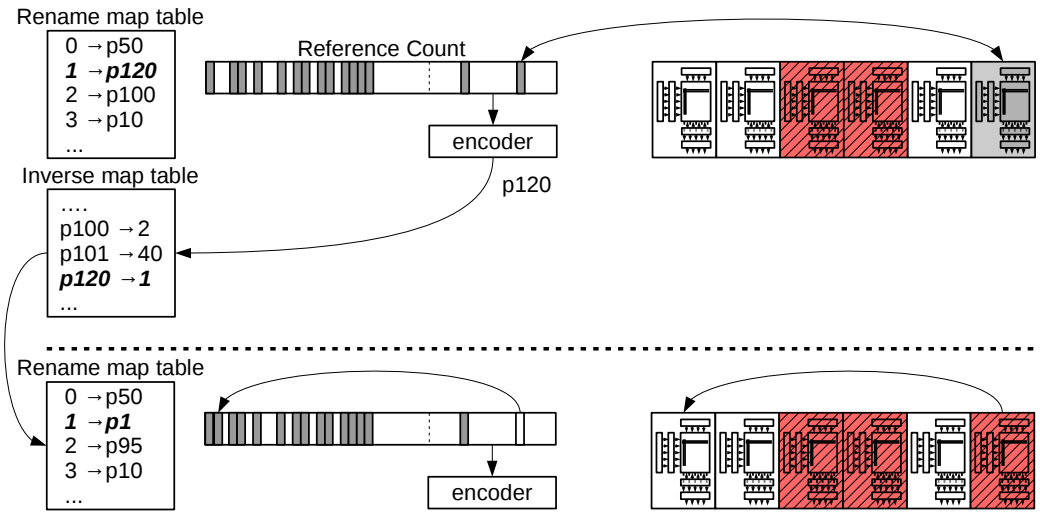
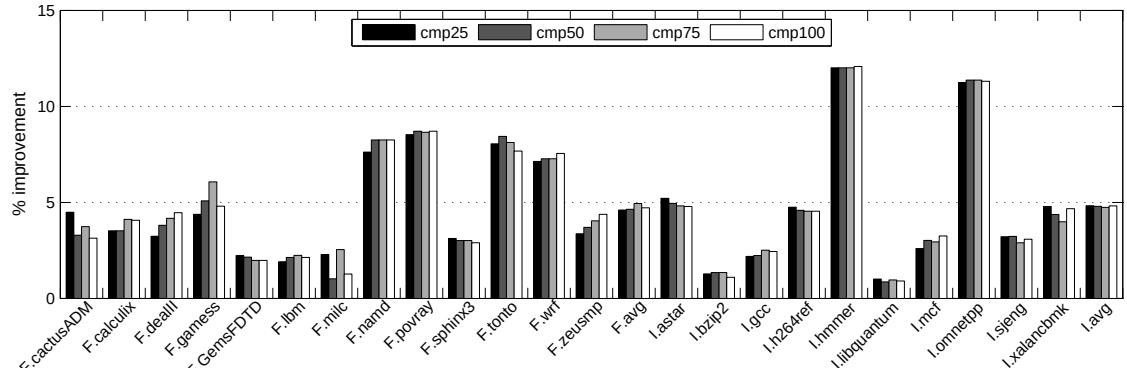


Figure 4.17: Register compaction mechanism. Disabled register file banks are shaded with diagonal lines. Top—a candidate preg is selected for compaction. Bottom—the preg has been moved and the rightmost register bank is power-gated.

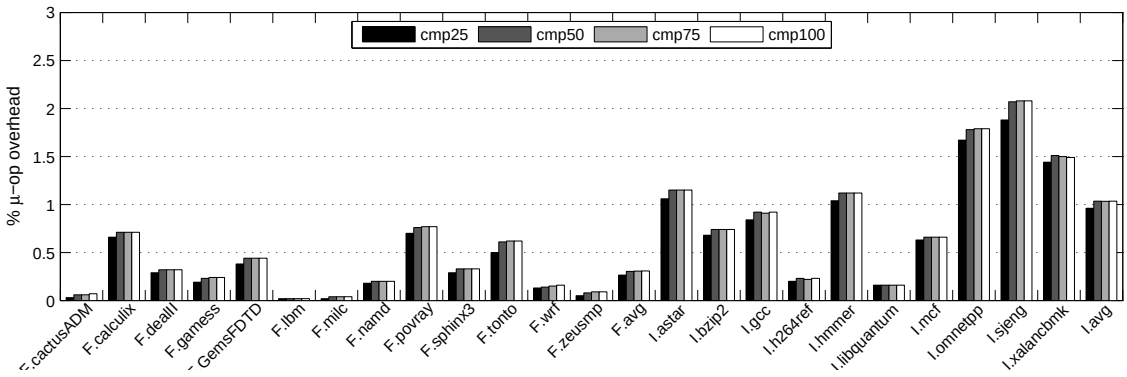
where a load instruction for example cannot overwrite the 'stuck' architectural register until the cache miss returns. To reduce the effect of these *lone* registers, the processor can dispatch a *compaction move* operation. This architecturally moves the logical register to itself, but micro-architecturally releases the 'stuck' physical register. The register value is effectively moved from the 'lone' bank to a lower order physical register.

Implementing compaction moves require maintaining an inverse rename map table. This table is similar to the traditional rename map table which maps logical registers to a larger pool of physical registers, only it performs the inverse operation, mapping physical registers to logical registers. A large physical to logical map table will be sparsely filled, as there can be many more physical registers than logical registers. To reduce the cost of this table, the table is limited to only the 64 highest numbered registers yielding a table of 64 physical registers mapping to 64 logical registers. These 64 physical registers are compaction source candidates for being moved from the high end of the register file to the lower end.

Figure 4.17 shows the mechanism for generating a compaction move operation. To generate a compaction move, a 64-bit wide priority encoder reads the reference count bitvector containing the



(a) Improvement in gating vs. no compaction moves



(b) Instruction overhead

Figure 4.18: Register compaction performance. Sweeping the ROB occupancy threshold at which moves are injected, from less than 25% full (cmp25) to injecting if the ROB has any slots free (cmp100) for banks of 8-registers, priority allocation and wm8 gating

candidate physical registers. In the figure, there are two candidates for compaction, shown to the right of the dashed line in the reference count vector. This register number is used to index into the inverse map table to identify the corresponding logical register. In the example above, physical register 120 is selected. The inverse map table shows this register is currently mapped to logical register 1. This micro-op is allocated a new, *lower-order* physical register and remaps the logical register to this new location. This micro-op updates the register reference counts and the rename map table, routing subsequent readers to the new physical register and injecting a compaction move micro-op into the dispatch queue.

There are several requirements which must be followed when injecting the instruction into the micro-op instruction stream. First, the instruction cannot be injected within another macro-op.

It must be placed *between* micro-ops. This is because instructions are committed on macro-op boundaries. There must also be enough bandwidth in the rename stage to perform the compaction move allocation, otherwise all ports into the rename map table will be busy. To reduce overhead, only one compaction move is allowed in flight at a time.

Compaction Evaluation

Figure 4.18a and b shows the effect of introducing compaction move operations into the instruction stream. We sweep the threshold required before compaction moves are introduced from 25% of the ROB being availble to 100% of the ROB available (compaction injected without restriction). In Figure 4.18a, we show the improvement in register file gating over the default priority-encoded allocation. This is the additional amount of the register file which can be power gated due to compacted registers. Power gating opportunity in both floating-point and fixed-point workloads improves by approximately 5%, and up to 10% for some integer workloads. In Figure 4.18b, we show the micro-op execution overhead due to the additional move operations introduced into the pipeline. Integer workloads incur an 1% increase in execution, while floating-point had fewer than 0.3% overhead.

There are fewer opportunities to compact registers when the ROB is full (i.e. fewer than 25% entries are available), causing a slight reduction in instruction overhead compared to the other ROB thresholds. There is little additional opportunity to improve register file gating due to the natural register steering from the priority allocation algorithm. With each compaction threshold, there is little change to both register opportunity and micro-op overhead, as all the opportunity is covered with the first 25% threshold. There is no benefit to relaxing the threshold for compaction. The micro-op execution overhead is due to the additional move instructions injected into the pipeline. This compaction move causes two registers to be occupied at once for the same logical register: the new destination along with the prior destination. These registers will both be occupied until the instruction is committed. A larger instruction window can cause these registers to be occupied for a long time until the younger destination is committed and the register freed. This dynamic micro-op cost effectively cancels out the static energy savings due to register file compaction.

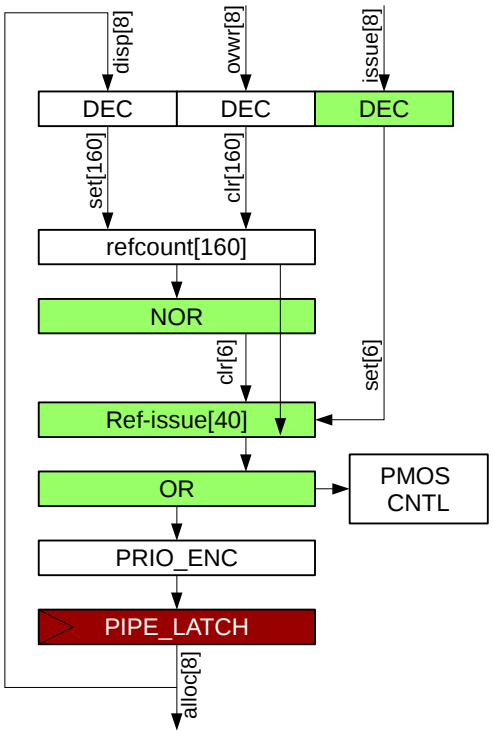


Figure 4.19: Reference counting mechanism for enabling banks at instruction issue.

4.8.2 Enable at Issue

Another way to improve leakage reduction is to absorb the allocate-write distance by keeping banks disabled until an instruction in the bank has issued to an execution unit. To achieve this, we add a bitvector recording when a register writing instruction has *issued*. An *issue* vector, shown in Figure 4.19, is fed by the reference count *allocation* bit-vector and drives the power gating control logic. This issue vector is much narrower than the allocation vector, requiring only a single-bit per bank. When an instruction issues, the physical register tag is decoded and sets a bit corresponding to the register *bank*.

The operation of this *issue* vector is as follows. 1. When the bank has no registers allocated, the issue-bit is cleared. 2. When an instruction is allocated a register from the bank and is dispatched into the issue queue, the bit remains the same. 3. When the instruction operands are ready and the instruction issues to an execution unit, the bit is set. This bit clears the vdd-gate signal, turning on

the bank. This allows the bank to be disabled immediately (if necessary) when the bank is empty, but only enabled when at least one of the instructions from that bank issues.

Figure 4.20 shows the improvement in power-gating performance for this bank-issue mechanism. Register banks are disabled slightly longer, and breakeven much more frequently. The total leakage energy per cycle is reduced by 0.5% for floating-point workloads and 3% for integer workloads. On average, register-file leakage energy is reduced by 4% for floating point workloads and 37% for integer workloads for banks of eight registers. Leakage energy for floating point workloads is not reduced more because the improvement in gating is not significant. Leakage per cycle is still dominated by the fraction of time the bank is enabled.

4.9 Energy Reduction

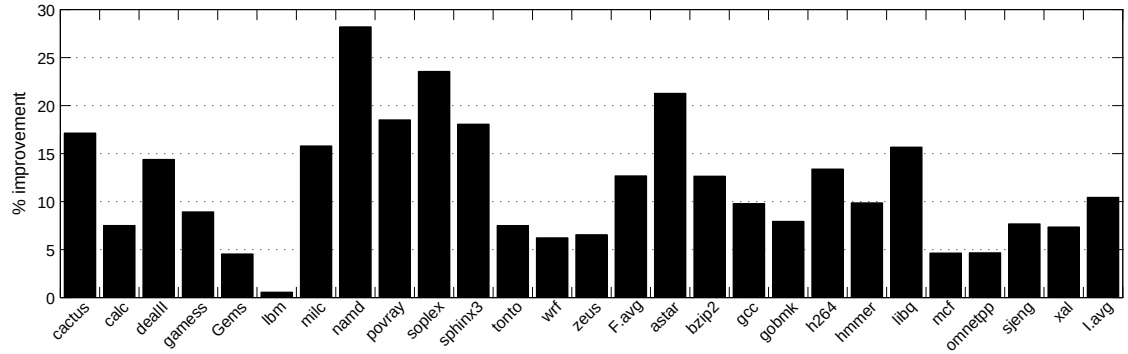
Although reference counting has a small energy (and area) overhead relative to a traditional free list implementation, the techniques it enables conserve enough energy to turn it into a net energy reduction technique. While we have detailed models of the register file and reference counting logic, we don't have similar models for the rest of the processor and therefore must make some assumptions. Specifically, we use published data from the POWER7 microprocessor [96] to map our simulation results to a processor power model. In the POWER7, the register file accounts for 20% of total core power—specifically, 14% of its leakage power and 24% of its dynamic power. Also, the POWER7 register file dynamic power is three times higher than register file leakage power. Given these baseline relationships, we can compute total energy as a function of static and dynamic energy:

$$E_{base} = E_{\text{static\_base}} + E_{\text{dynamic\_base}} \tag{4.2}$$

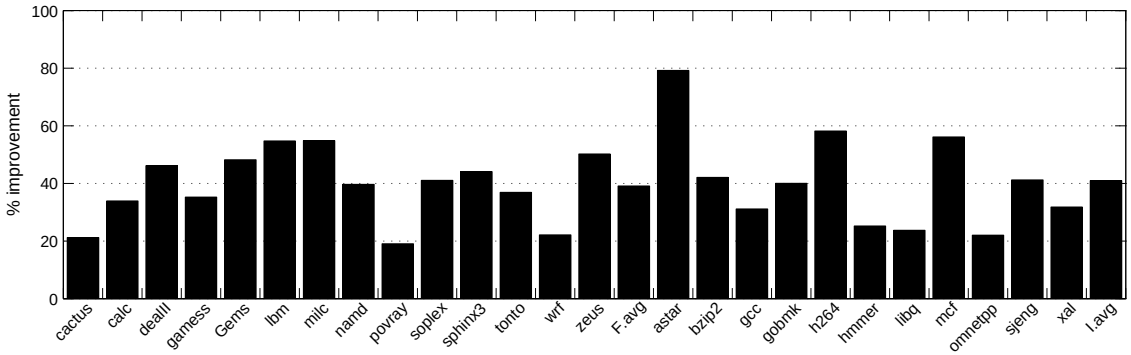
Static energy is calculated using the relationship of register file leakage to total leakage from POWER7.

$$E_{\text{static\_base}} = \frac{E_{\text{static\_base\_RF}}}{0.14} \tag{4.3}$$

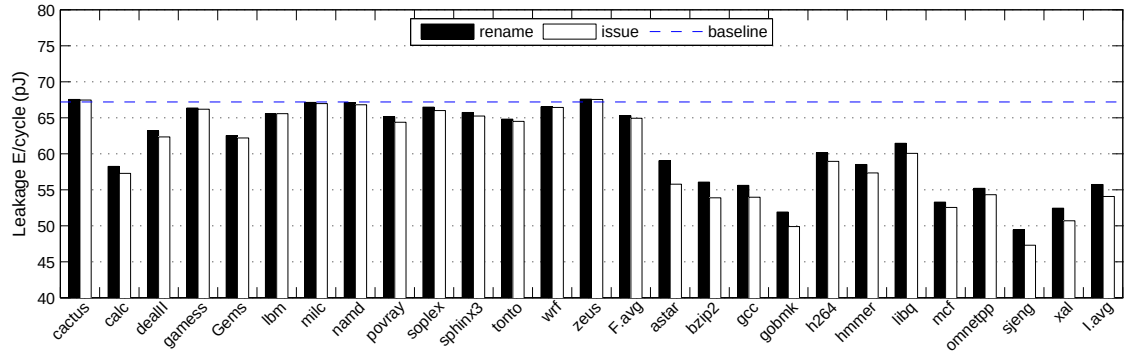
$$E_{\text{static\_base\_nonRF}} = E_{\text{static\_base}} - E_{\text{static\_base\_RF}} \tag{4.4}$$



(a) Improvement in gating by enabling at issue



(b) Improvement in breakeven ratio by enabling at issue



(c) Leakage energy per cycle vs enable at rename

Figure 4.20: Enable at issue for banks of eight-registers allocated with priority algorithm, watermarked gating

We calculate dynamic energy with the following,scaling the baseline register file static energy:

$$E_{\text{dynamic\_base}} = E_{\text{static\_base\_RF}} \times \frac{3}{0.24} \tag{4.5}$$

$$E_{\text{dynamic\_base\_RF}} = E_{\text{dynamic\_base}} \times 0.24 \tag{4.6}$$

$$E_{\text{dynamic\_base\_nonRF}} = E_{\text{dynamic\_base}} - E_{\text{dynamic\_base\_RF}} \tag{4.7}$$

For a non-baseline processor with reference counting, we calculate static energy using the baseline calculated above.

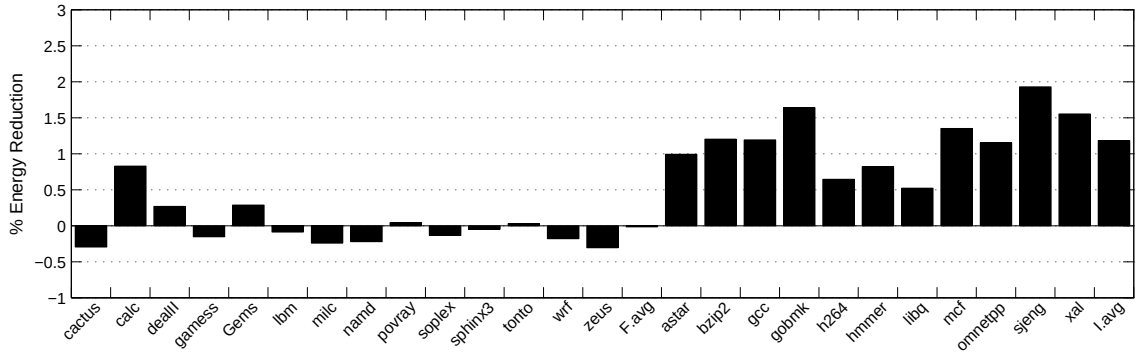
$$E_{static} = \left(E_{\text{static\_base\_nonRF}} + \frac{P_{\text{static\_RF}}}{P_{\text{static\_base\_RF}}} \times E_{\text{static\_base\_RF}} \right) \times \frac{CC}{CC_{base}} \tag{4.8}$$

P is power and CC is cycle count. Essentially, we leave the non-register file portion untouched and scale the register file portion using the circuit modeled ratio of old to new register file—here 'register file' includes register management structures, free list or reference counting. We then scale by relative execution time (cycle count). We apply the same technique to dynamic power, scaling this time by relative instruction execution count (IC). We discount the execution of injected compaction moves by a factor of 0.5 because compaction does not change fetch or decode pipeline stage activity.

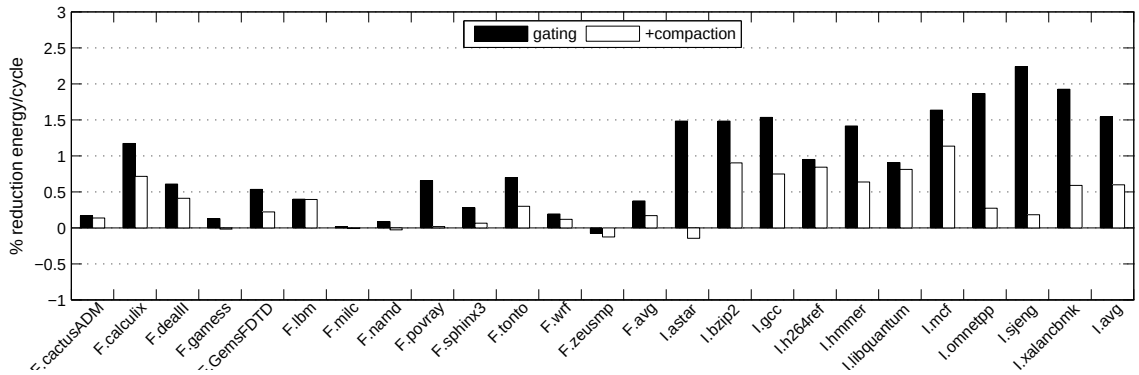
$$E_{dyn} = \left(E_{\text{dyn\_base\_nonRF}} + \frac{P_{\text{dyn\_RF}}}{P_{\text{dyn\_base\_RF}}} \times E_{\text{dyn\_base\_RF}} \right) \times \left(\frac{IC + 0.5 \times IC_{\text{compact\_move}}}{IC_{base}} \right) \tag{4.9}$$

Figure 4.21 shows the effect in energy delay from implementing flexible register management structures and enabling register file power gating. In Figure 4.21a, we see the change in total system energy when applying register file power gating and enabling banks when an instruction issues. The instruction count and cycle count remain constant, as the only change is the allocation and gating algorithm. Enabling at issue reduces the leakage cost of the

Figure 4.21b shows vdd-gating (black) and compaction moves (white) for banks of *four* registers. This figure shows that inserting compaction moves is a net energy sink, as each compaction bar is lower than the corresponding vdd-gating bar. The dynamic costs of inserting the compaction



(a) Total energy reduction: enable at issue vs. clock gating, banks of eight-registers



(b) Total energy reduction: compaction moves and vdd-gating gating vs. clock-gating, banks of four-registers

Figure 4.21: Change in system energy

move instruction eliminates not only any improvements that could result from power-gating more of the register file, but consumes energy that was recovered by the default power-gating configuration. There is also a slight increase in register pressure and execution time for compaction moves because while the compaction move is in flight, the logical register will effectively consume two physical registers. Compaction moves effectively increase energy consumption and increase execution time compared to vanilla power-gating.

In order to achieve more significant energy reduction, register occupancy needs to be reduced. This can be achieved through through speculative retirement and execution driven register reclamation and by using register sharing [4].

Chapter 5: Scalable Micro-architecture

Latency tolerant architectures promise significant performance improvements for out-of-order cores when faced with long-latency memory accesses. Latency tolerant architectures expand the instruction scheduling window by removing instructions dependent on the miss from the critical path, and reinserting them into the pipeline when the missing data returns from memory. This approach increases instruction-level parallelism (ILP) and exposes memory-level parallelism (MLP) by overlapping more cache misses. However, there is a cost to current approaches to latency-tolerance; these architectures required expensive value-copy operations to shuffle data from the register-files into the slice buffers or ROB, in contrast to the pointer-based register mapping paradigm used by many out-of-order cores. These architectures can also execute a significant number of extra instructions by deferring loads that would have been squashed or speculating too deeply beyond branches, flushing back to checkpoints that could be hundreds or thousands of instructions younger. Thus, for mobile and energy-constrained applications, current latency-tolerant architectures are not attractive.

In this chapter, we describe a scalable and efficient microarchitecture to address the energy and execution inefficiencies of existing latency tolerant architectures. The SCREW (Scaling Resources in Efficient Ways) microarchitecture predicts which loads will miss in the LLC and steers miss-dependent instructions to FIFO instruction-buffers. This simple FIFO scheduling allows SCREW to scale the issue queue and reduce expensive issue-queue wakeup broadcasts. We introduce techniques for tracking dependencies and scaling the register file and ROB by applying speculative retirement.

The SCREW microarchitecture performs *pre-execution filtering*, deferring instructions dependent on a load predicted to miss the cache at the *front-end* of the processor, before the instructions enter the out-of-order execution pipeline. This mechanism prevents dependent instructions from occupying critical issue-queue resources which can be more efficiently used by instructions *independent* of cache misses. *Speculative retirement* allows SCREW to scale the *back-end* of the pipeline when a cache-miss is blocking the processor from making forward-progress. This speculative retirement mechanism

(described later) allows instructions to make speculative progress into a checkpoint while the cache-miss is outstanding.

The goal of the SCREW microarchitecture is not only to scale all structures which constrain the instruction window, but do so efficiently, with low-complexity and low-energy structures. By adding relatively simple lookup-tables and FIFO queues to the front end of the pipeline, SCREW is able to make better use of the out-of-order scheduling resources, which include the register file, issue queue, and re-order buffer.

SCREW adds a FIFO queue as a new destination to buffer instructions waiting to be scheduled. The FIFO queue stores only those instructions dependent on loads predicted to miss the cache. These instructions will issue serially (*i.e.* in FIFO order) when the load miss returns. Unlike the conventional out-of-order issue-queue, the in-order FIFO queue does not perform expensive comparison checks against every entry. In an issue queue, each instruction must compare its un-ready operands against each writeback broadcast from younger instructions. This requires expensive writeback buses, comparators, and buffers. These costs prevent issue queues from scaling effectively to hold more instructions [64, 52]. FIFO energy and area efficiency comes at the expensive of performance, as instructions are serialized in a FIFO rather than woken up out-of-order. However, performance loss is mitigated as the issue queue is no longer blocked with miss-dependent instructions who often do not benefit from out-of-order scheduling.

Unlike slice-buffers proposed in previous designs such as CFP [82] and BOLT [35], SCREW's FIFO queue design does not require expensive register copy operations to store data, nor does it perform searches to determine which instructions are ready for slice-in. This queue is a compact and efficient design requiring only register identifier pointers and incrementers. In this microarchitecture, data is kept in the register file and is not copied to slice buffers or stored in an expanded ROB. This streamlines the process, as expensive data buses are not required between the ROB, register file, and scheduling queues. Retirement map checkpoints and reference counting register management described in Chapter 3 enable speculative retirement. This allows SCREW to recover reorder buffer and register-file resources from instructions who have completed their execution at the tail end of

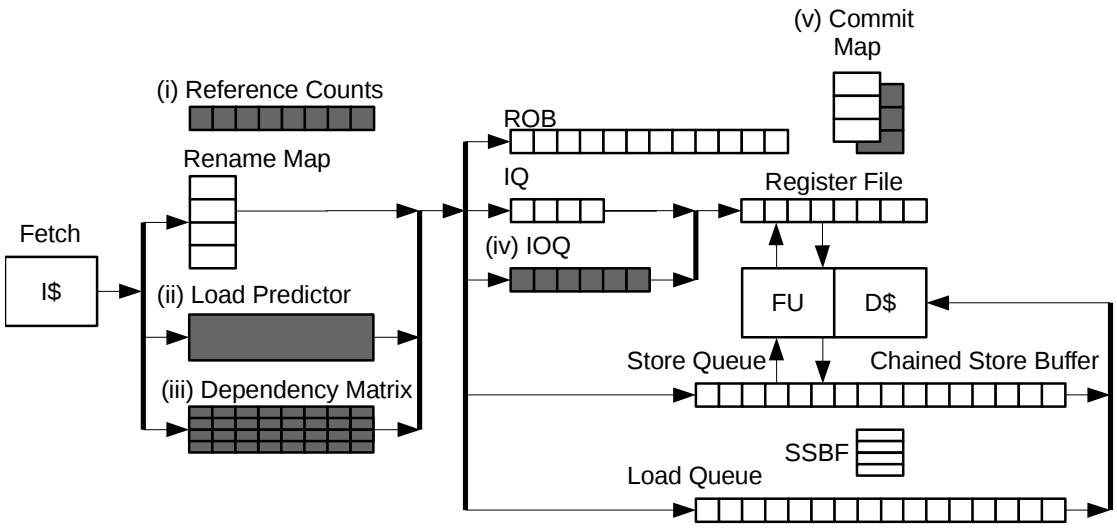


Figure 5.1: SCREW block diagram. New SCREW structures are highlighted

the pipeline earlier than in a conventional out-of-order processor. These structures unlock more performance in the processor, allowing the scheduler to execute around long latency cache misses, increasing instruction throughput and workload performance.

5.1 SCREW Overview

The SCREW microarchitecture, shown in Figure 5.1, is an evolution from prior latency tolerant designs such as BOLT and CFP. The focus of SCREW is scheduling instructions dependent on a long-latency cache miss from FIFO in-order queue(s). The microarchitecture is optimized around these FIFO structures to improve performance and provide energy-efficient latency tolerance. Prior LT microarchitectures such as WIB, BOLT, and CFP have shown that a FIFO is the ideal scheduling mechanism for miss-dependent instructions, rather than a CAM-based issue queue [45, 35, 82]. SCREW uses a novel predictive technique to steer instructions into FIFO queues at the front-end of the pipeline, rather than a"dispatch/wake-up/pseudo-execute/copy-to-FIFO" mechanism as in BOLT and CFP. In previous microarchitectures, the latency tolerant mechanism attempts to make a correction. BOLT and CFP shift the instructions from the expensive issue queue CAM to a cheaper FIFO, while WIB stores instructions in a both a FIFO queue and the issue queue. In SCREW, the

latency dependent instructions are dispatched to the cheaper FIFO structure by design and do not need to "slice out" from the out-of-order pipeline. This corrective procedure in prior designs costs energy to copy the instruction and operands out from the issue queue, register files and into slice buffers.

The SCREW microarchitecture must support several functions in order to scale the out-of-order window. First, SCREW must be able to both identify loads which will likely miss the cache and track dependencies on these instructions, creating load-dependency chains. Once SCREW has identified long-latency loads and captured the dependent instructions, SCREW can scale the re-order buffer and register file by performing speculative retirement. Speculative retirement requires several actions to be supported by the microarchitecture. The first task is to checkpoint the committed register state as a 'save-point' to restore in case there is an exception during the checkpoint. This requires the microarchitecture to support register pinning, preventing checkpointed registers from being freed and overwritten, removing them from the free list. Finally, new scheduling and commit mechanisms are required to remove instructions from the ROB (earlier than in conventional processors for miss-dependent instructions) reclaim registers after execution rather than at commit-time, and commit the checkpoints once all instructions have written back.

SCREW augments the conventional out-of-order processor pipeline with new structures shown in Figure 5.1 to support these new functions. These structures include: (i) register reference counts to manage register allocation for conventional retirement and execution-driven reclamation (to scale the RF) during speculative retirement; (ii) a long-latency load predictor to predict which load instructions will miss the cache; (iii) a bit-matrix identifying the destination registers of instructions dependent on the load instruction; (iv) the *in-order queue* FIFO buffer(s) which contain the sliced instructions; and (v) the commit-map checkpoints to support speculative retirement and scale the ROB. SCREW leverages existing mechanisms to scale the load and store queues to support large instruction windows: notably speculatively indexed (non-associative) store queues [78], store-vulnerability window [72] extended with decoupled store completion with silent deterministic replay (DSC-SDR) [34] to ensure memory consistency within the checkpoint, and the chained-store buffer

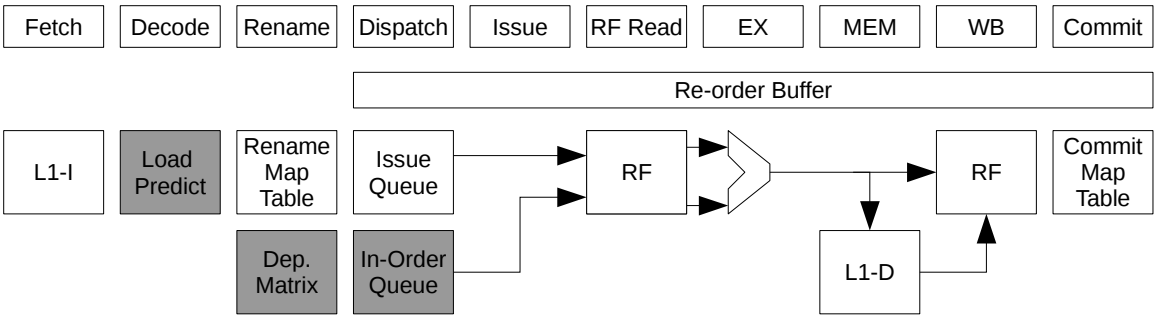


Figure 5.2: SCREW pipeline. New SCREW structures are highlighted

concept [33] to forward data from stores to "deferred" loads.

Figure 5.2 shows a high-level overview of a SCREW pipeline. The pieces of the pipeline modified to support issue queue scaling are shaded in gray. During decode, loads are predicted to either hit or miss the last level cache. During rename, a dependency chain is created for the each load (predicted to hit or miss). Each instruction searches this matrix to identify the loads it depends on. Identifying the loads on which each instructions depends on (either directly or through a chain of dependencies) is the key that allows SCREW to manipulate the out-of-order instruction window. Instructions in a chain belonging to a predicted cache-miss are steered to the FIFO queue. Instructions steered to the FIFO are issued in-order when the load-miss they depend on returns. Predicted miss-independent instructions are dispatched normally into the issue queue.

5.1.1 Comparison to Prior LT Designs

SCREW builds upon several prior LT microarchitectures, with an emphasis on energy efficient slice-out. At a high level, SCREW is similar to WIB in that instructions are allocated a slot in the slice-buffer at dispatch in *program-order*. This is in contrast to architectures like CFP and BOLT which pseudo-execute instructions through the pipeline to release scheduling resources in *execution-order* (*i.e.* after dependents wake up) before slicing out from the ROB in *program-order*.

Compared to WIB, SCREW's slice-out and execution mechanisms are much more efficient. In a WIB processor, instructions are issued only from the issue queue but are stored in both the WIB (slice-buffer) *and* issue queue. An instruction in WIB is "sliced" and removed from the issue queue

if it depends on a load-miss. The issue queue is informed of the cache-miss after it occurs, and the instructions waiting on the load destination register are removed from the queue. These instructions are re-injected into the issue queue from the "waiting instruction buffer" when the load data returns. This mechanism is inefficient at dealing with chains of loads. A stream of instructions can ping-pong between the WIB and issue-queue if a series of loads are sliced out where loads in the dependency chain miss in the cache. In SCREW, instructions are dispatched into only one structure—the issue queue for loads predicted to hit in the cache and their dependent instructions, and the in-order queue for instructions depending on loads predicted to miss the cache. Unlike WIB, BOLT, and CFP, instructions can only issue from the structure they were dispatched into; there is no duplication of instruction data between multiple structures, no thrashing between queues where instructions are re-injected and re-sliced, and no redundant dispatch or issue.

At the tail end of the pipeline, SCREW is conceptually similar to a BOLT [35] and CPR [2] processor. When the head of the ROB is blocked from retiring because it is a load instruction waiting for a cache miss to return, a BOLT processor will create a checkpoint and speculatively retire "poisoned" instructions which depend on the miss. SCREW performs the same actions, but has a different organization and slice-out flow. In BOLT, instructions dependent on the load-miss are identified through "poison" bits which propagate from the load missing at the head of the ROB backwards to dependent instructions in the issue-queue. Poisoned instructions are removed from the head of the ROB and issue-queue, and are placed instead into a "slice-buffer" which functions as a secondary ROB. This slice-buffer ROB looks similar to the ROB style found in older Pentium processors—instead of storing only instruction meta-data and register pointers, the BOLT slice-buffer ROB stores both the instruction *and* the data value of a ready input operand. This allows BOLT to free registers that are consumed by poisoned instructions at the cost of a larger ROB and the cost of transferring the data.

SCREW enters a speculative retirement regime when the ROB is blocked by a load miss with a similar checkpointing mechanism. SCREW has identified loads likely to miss and has already tracked their dependents, so no 'slicing' or transferring is necessary. The dependency matrix also

tracks dependency chains for loads predicted to hit. If this prediction is incorrect, SCREW can still checkpoint the register state and enter a speculative-retirement mode because each instruction is tagged with the load chains they depend on. An instruction dependent on a load incorrectly predicted to hit can be added to a checkpoint *prior* to finishing execution. Instructions independent of the load miss will not be added to the checkpoint until they finish execution normally.

Unlike BOLT and CFP, SCREW does not need to copy instructions dependent on the load-miss from the ROB into the slice-buffer, as they will have already been placed there at dispatch due to the load prediction mechanism at the front-end of the pipeline. SCREWs speculative-retirement procedure also requires less energy than in BOLT and CFP. SCREW only needs to remove a sliced instructions instructions entry from the re-order buffer when speculatively retiring it; SCREW does not need to perform a value copy because it does not release these registers, nor does SCREW need to propagate poison, because the dependency chains are tracked at the front-end of the pipeline. SCREW's FIFO queue more closely resembles modern re-order buffers with register pointers, rather than an older Pentium style buffer which stores the full data value [57]. This organization eliminates a register-read and value-copy operation, at the cost of potential decreased performance, as destination registers are kept occupied while instructions are waiting in the FIFO buffer for their load to return. While this does prevent registers from being reused for 'active' instructions, rather than 'waiting' instructions, this scheme does lend itself well to power-gated and drowsy modes for registers described earlier in Chapter 4.

SCREW reduces the energy cost of latency tolerance by replacing a reactive slice-out approach—poison-execution/value copy/re-dispatch/re-issue—in CFP and BOLT with pro-active dispatch approach. SCREW steers instructions likely to be miss-dependent into a FIFO, avoiding the issue queue entirely. SCREW also takes a different approach to scaling the physical register file. SCREW's speculative retirement is re-designed to release the physical registers of miss-independent instructions and eliminates the value-copy operations prevalent in BOLT. There is no re-renaming or re-dispatching into the issue queue after the load returns, saving energy and reducing complexity.

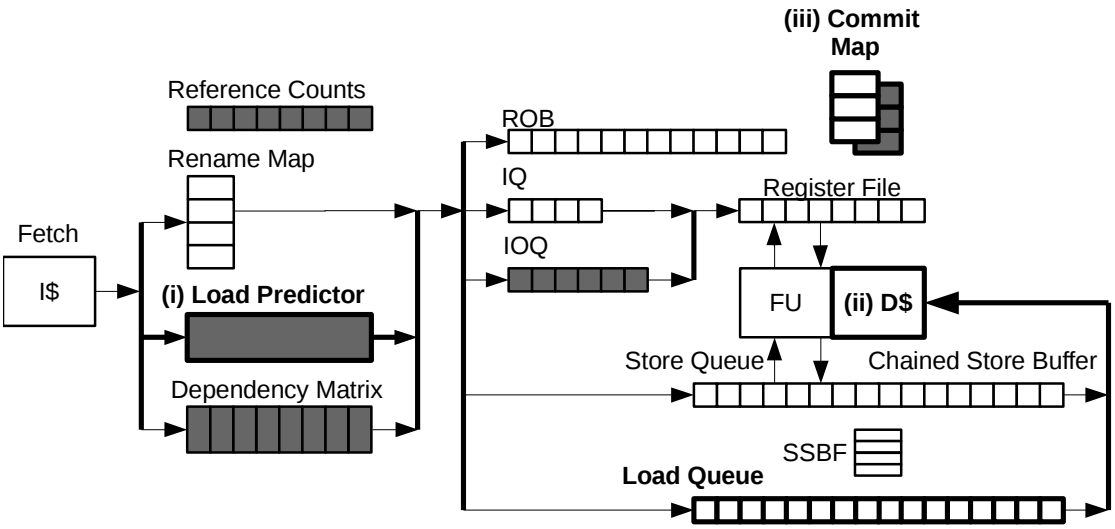


Figure 5.3: SCREW block diagram with load prediction structures highlighted.

5.2 Load-Miss Prediction

Load behaviour prediction is a pre-requisite for steering instructions at dispatch; the SCREW processor cannot wait until a load has executed in order to capture its cache behaviour. The pipeline shown in Figure 5.2 shows the load behaviour predictor at the front-end of the pipeline in the decode stages. Once the instruction is classified as a load, it can index into the load-miss predictor using the instruction address to get a prediction about its cache behaviour. All load instructions access this predictor in order to determine the behaviour of younger dependent instructions. The prediction determines if its dependent instructions—those instructions reading the load instructions destination register—must be diverted to the in-order queues. The load proceeds to be dispatched to the issue queue if it is not dependent on any prior predicted load misses.

Branch prediction techniques have been applied to load instructions for improving cache performance [89, 46, 91] and instruction scheduling [41, 54, 94]. Caches benefit from knowing whether or not a load will be re-referenced, preventing the cache from being polluted with useless data. A prediction is made to determine if data that is brought into the cache should be allocated and kept for future references. Schedulers benefit from knowing whether or not a load will have a short or

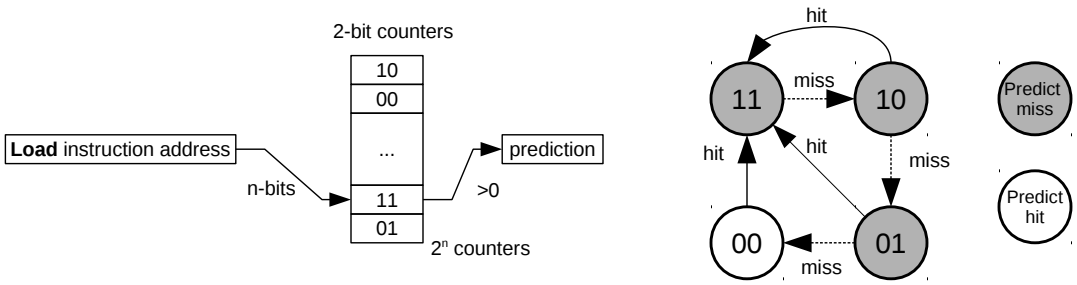


Figure 5.4: left: Local predictor indexed by n-bits of PC. Right: counter state machine. Counter saturates with hit, decrements with miss. >0 predicts miss.

long latency. If a load is known to have a short latency, its dependent instructions can be issued assuming a fixed load-latency, rather than waiting for an indication that the load data has been accessed. In SCREW, we apply a prediction for what type of scheduling an instruction needs. If the instruction is dependent on a load-hit, it will need high performance out-of-order scheduling because the load will have a fixed-short latency. Instructions dependent on a load-miss can use cheaper, in-order scheduling because the load miss will have the largest effect on latency.

Figure 5.3 shows the structures involved in load-miss prediction. The first structure is the load predictor which makes the cache-behaviour prediction; the load-queue and data cache perform the access and record a hit or a miss; finally at commit, the predictor is updated with the behaviour of the predictor. Load-access predictions can be made prior to dispatch, during any of the decode or register rename pipeline cycles. Once the instruction type is known to be a load, the instruction address can be used to index into the predictor as shown in Figure 5.4. This figure shows a simple 'local history' predictor, where each counter is accessed by only the instruction address. The predictor counters predict a hit if the value is non-zero. The value saturate when updated after a cache miss. This requires 3 consecutive hits to the same load in order for the load to predict a hit after a miss.

5.3 Execution Flow

Figure 5.5 shows a stream of instructions with four loads highlighted in **bold**. The first load is predicted to hit the cache and is dispatched to the issue queue. The subsequent three loads are predicted to miss. Instructions dependent on these loads will be identified and steered to the FIFO

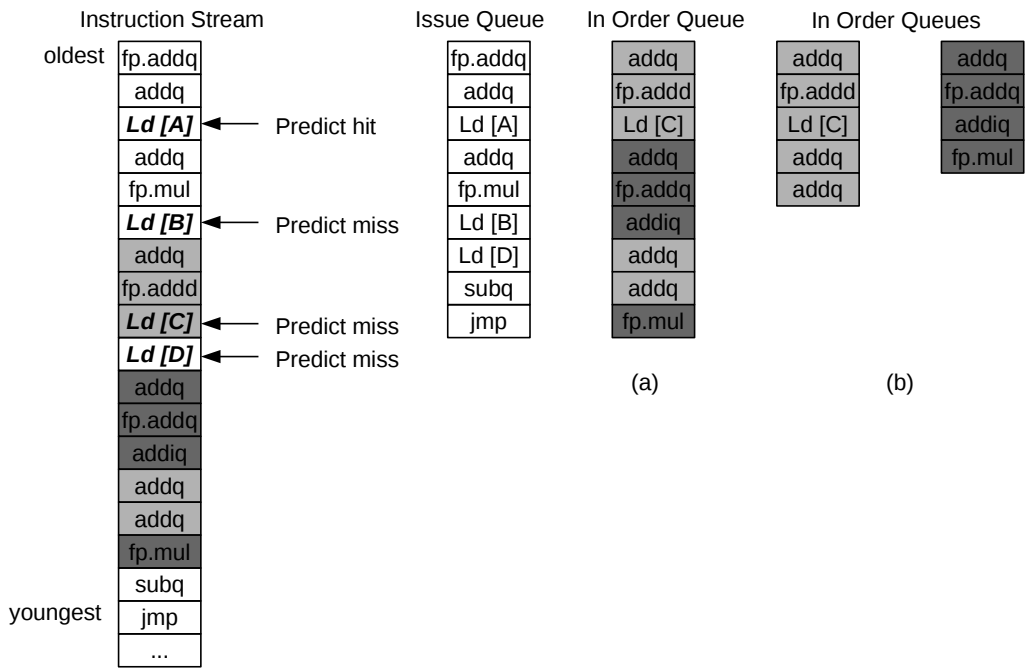


Figure 5.5: Prediction and steering of load-miss dependent instructions to (a) single FIFO queue and (b) multiple FIFO queues

in-order queue. Note that in this example, `load[c]` is dependent on `load[b]` and will be steered to the FIFO queue instead of the issue queue. `Load[b]` is not dependent on a predicted load-miss and will be steered to the issue queue as with other miss-independent instructions. `Load[d]` is independent of these other loads. It is predicted to miss, with its younger dependents highlighted in dark grey.

Instructions dependent on the long latency load are identified at rename using a dependency matrix. Each instruction will check this dependency bit matrix to identify if any input logical registers are written by long-latency-dependent instructions. If a register source is found in this matrix, the instruction will be steered to an in-order-queue, *i.e.* the instruction is sliced out from the short latency instruction stream.

As part of this process, the instruction will add its logical destination register to the matrix so any new instruction reading that register will also be sliced out to the FIFO queue. From this point forward, sliced instructions are treated differently than conventional miss-independent instructions.

Dependent instructions are dispatched to the FIFO queue, rather than the issue queue. These dependent instructions can remain live for hundreds of cycles if the load instruction must access main memory, and would otherwise occupy issue-queue slots which prevents younger instructions from entering the pipeline.

In Figure 5.5(a), only one FIFO queue is available for miss-dependent instructions. This queue co-mingles instructions dependent on multiple loads. In Figure 5.5(b), multiple FIFO queues are available for miss-dependent instructions. Each load chain is assigned to a different queue, keeping chains of instructions in independent queues. This prevents one chain of instructions from blocking another chain from making forward progress.

An instruction at the head of the FIFO queue will check the register scoreboard to identify if its register operands are ready. These FIFO scheduled instructions do not need to broadcast their results, as the execution latency will be known for non load instructions. This is in contrast to the conventional issue-queue scheduling mechanism, where *every* instruction will broadcast its destination register tag to the issue-queue in the writeback stage. When the operands are ready, the instruction issues from the FIFO queue to an execution unit and the head pointer is incremented, 'popping' the instruction from the queue. At this point the miss will have returned—otherwise the instruction could not be scheduled—and the instruction proceeds through the pipeline. This is in contrast to the WIB design, which re-inserts instructions from the WIB back into the issue queue after the load returns, blocking younger instructions from dispatching. This allows for out-of-order scheduling at the cost of copying data back and forth between buffers and potentially thrashing between the two structures.

The queue and predictor mechanisms effectively serializes streams of instructions. The instruction schedule is locked in program order, and ILP can only be extracted by overlapping instruction latencies, rather than through scheduling. Any performance loss due to this 'serialization' is overcome by the additional ILP gains from scheduling more instructions from the issue queue. Multiple FIFO queues allow chains of dependent instructions to be overlapped.

At the tail end of the pipeline, instructions retire when they are the head of the ROB. If the

ROB head is a load, then the microarchitecture has tracked all of the dependent instructions and associated them with that load. The processor will know that if the dependent instructions are stored in the FIFO if the load is either (a) a load predicted to miss or (b) a load dependent on a load predicted to miss. If the instruction has finished execution and written its target register, it is retired as normal. If the load is stalled and blocking the pipeline because it has missed the cache, the SCREW processor checkpoints the architected state and enters a speculative-retirement mode to unclog the pipeline. Instructions are removed from the ROB early if they are depend on the miss, while miss-independent instructions are removed and checkpointed at writeback. These checkpoints can grow to hundreds and thousands of instructions as the long-latency miss can have a large 'shadow' of instructions dependent on the load. By removing the instructions from the ROB and aggressively reclaiming physical registers, more resources are available to allow younger instructions into the pipeline. Checkpoints are committed in order once all instructions in the checkpoint have executed and written their results back. We describe these mechanisms in detail in later sections.

5.4 Tracking Dependent Instructions

After SCREW predicts that a load will miss, it must propagate that information to instructions dependent on the load. Such a process is analogous to poison propagation in BOLT and CFP, but must be done prior to dispatch (*i.e.*, at rename), instead of during execution. The "obvious" approach here would be to add a "poison" bit to the rename map table, however, it turns out that the "obvious" approach needs a bit of augmentation.

The problem with this simple approach is that there is no easy way to clear the "poison" when a load miss returns, or when a predicted miss is discovered to actually be a hit. An idealized approach would be to make a dependency bit-matrix with one column per load, and one row per logical register, shown in Figure 5.6. Each column in the bit-matrix is a vector of 63 bits, where each row (bit) represents a logical register in the ISA. There are 63 usable logical registers described in the x86 ISA as register 64 is a non-writeable '0' register.

In such an approach, whenever an instruction is renamed, it reads the rows for each of its register sources. The instruction then ORs the rows from its register sources together, and writes them to

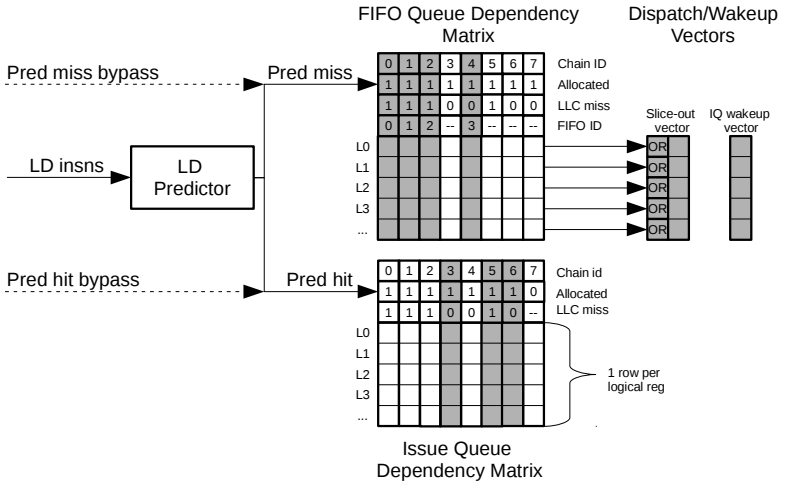
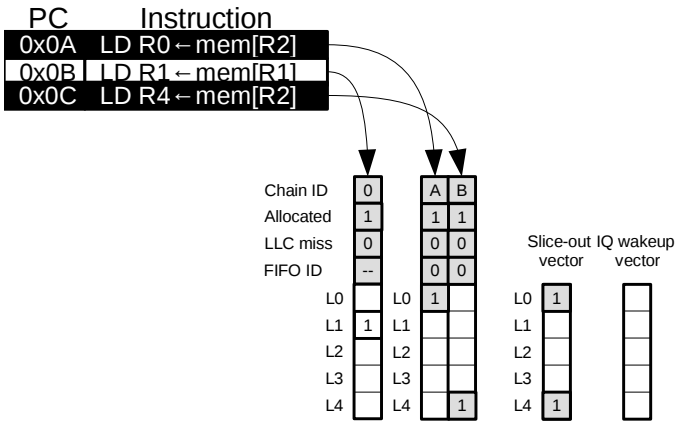
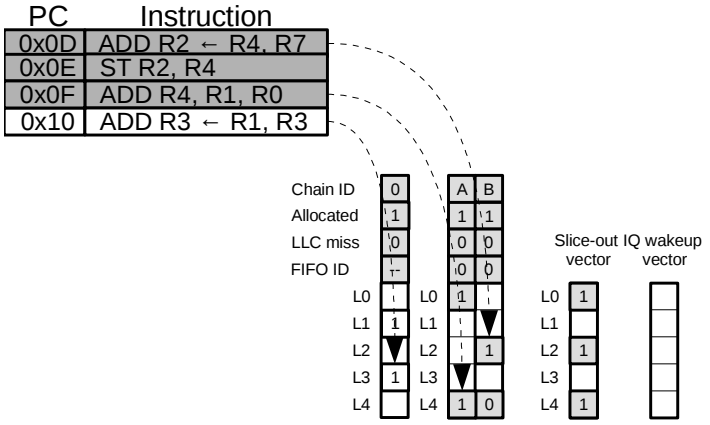


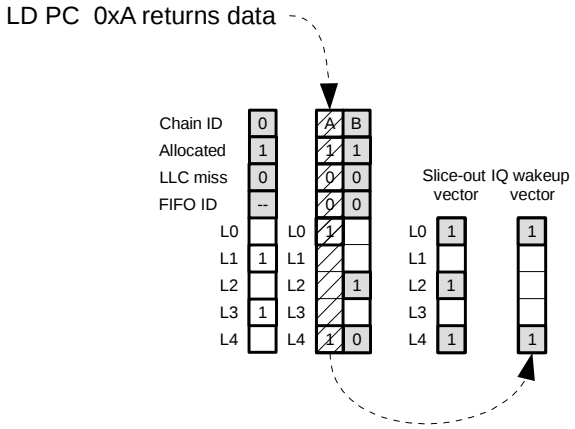
Figure 5.6: SCREW dependency matrix with interleaved dependency chains. Top: predicted-miss (FIFO) chains. Bottom: Predicted hit (IQ) chains.



(a) allocating a chain and setting bits

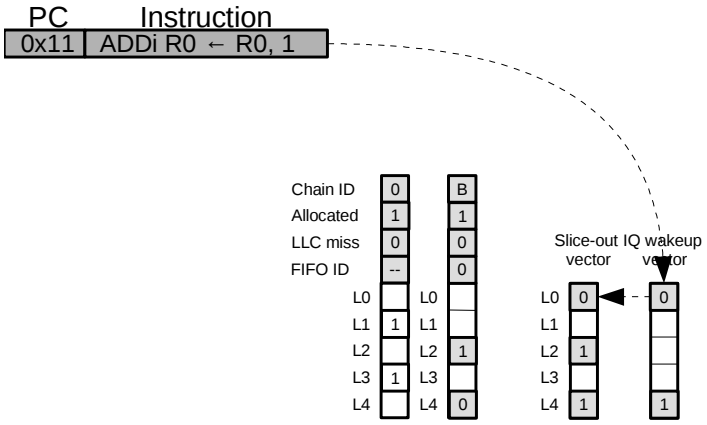


(b) clearing bits

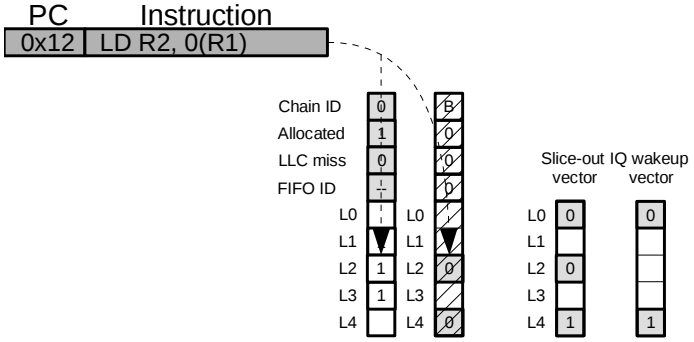


(c) clearing a chain when load data returns

Figure 5.7: Dependency tracking example.



(a) IQ wakeup insns



(b) ending a dependence chain

Figure 5.8: Dependency tracking example continued.

the row for its destination register. The instruction determines if it is dependent on a miss (and thus should be steered to the FIFO) by OR-reducing the bit-vector for its output register. Under such a scheme, the problem of what to do when a load miss returns, or a predicted miss turns out to be a hit, is solved by column-clearing the load's column in the matrix.

Of course, the problem with such an idealized design is that one column per load requires a rather large matrix (*e.g.*, one column per load queue entry, as that limits the number of in-flight loads). For an energy conscious design, such as SCREW, we would prefer to reduce the number of columns significantly (*e.g.*, to 8 or 16). New loads are allocated an "empty" column (one that is all 0s). Such a column is easy to detect, and columns will clear regularly, either as a result of loads hitting in the cache, or as a result of all of all of the registers in the chain being overwritten by younger, independent, instructions.

When all columns are occupied, SCREW cannot track the load. SCREW treats the load as if it were a typical logical instruction, adding its target register to any column it belongs to in the dependency matrix, and dispatching to a FIFO or out-of-order queue based on its membership in the existing load chains, rather than on its own predicted behaviour.

Each instruction is tagged with the chains it belongs to. This allows SCREW to quickly identify all dependents of an outstanding load miss. This is used for managing physical registers and quickly freeing ROB entries during speculative retirement.

The allocation procedure also sets the bit corresponding to the target register of the load —*e.g.* ld[R1] $\rightarrow$ R5 will set bit 5 in the column—poisoning that register for all younger instructions. When a younger load is decoded, it is allocated a new column and sets its destination register in the corresponding row. Any instruction whose sources are 'poisoned' by a long-latency chain must be steered to the FIFO. If the matrix is full when a load is decoded, then a new chain is not created. Dependencies are assigned to chains where the exist. If none exist, then the load is effectively not tracked and younger dependent instructions will behave as they would in a conventional processor.

5.4.1 Creating Dependencies

Figure 5.7a shows the mechanisms for allocating a chain and setting bits in the dependency matrix. Three loads are encountered and are each allocated a dependency chain. Load 0x0A and 0x0C are a predict 'miss' loads, and will set a chain whose dependent instructions will dispatch to the FIFO queue. Instruction 0x0A sets bit '0' in its chain and clears it from all other chains, which 0x0C sets bit '4' in its chain and clears it from all other chains. Load 0x0B is a predict 'hit' load, and will set a chain where dependent instructions dispatch to the issue queue. This instruction sets bit '1' in its chain and clears it from all other chains. The 'slice out' vector is the 'OR' or all the FIFO dependency chains, and will dispatch instructions with R0 and R4 sources. Instructions such as stores and conditional branches or jump instructions can be poisoned and steered to a FIFO without setting a bit in the vector.

When the matrix is full, a chain cannot be allocated to the new load. This load is effectively treated as if it were a scalar/logical instruction for dependency tracking, rather than a load instruction. The load clears a bit in all chains, indicating the new register mapping, but will not be assigned a new chain. If the load is dependent on other chains, it will set bits in those chains, but it will not control its own dependency chain.

5.4.2 Removing Dependencies

Figure 5.7b shows how bits are cleared When an instruction is miss-independent, it will have no 'poison' bits on its input logical registers and must clear the entire row for its destination logical register to prevent younger instructions from being steered to the FIFO incorrectly. Clearing the bit ends the dependency path through that particular destination register.

The second case where a bit is cleared is when an instruction *is* dependent on a predicted load-miss, but the output destination register is found in multiple vectors. If the source is only poisoned through a single vector, then the destination should only be set in that particular column of the bit matrix. In this case, the instruction clears the row and sets the bit only in the columns in which it receives poison.

The final case where a bit can be cleared is when the instruction producing the poison bit has

been squashed due to branch speculation. When a branch prediction is discovered to be incorrect, instructions younger than the branch are squashed. These instructions need to 'undo' any perturbations they made to the processor pipeline which includes register mappings and dependency bits. When instructions roll-back to the branch instruction PC, bits that were cleared by younger writers need to be reset, while bits that are set by younger,squashed instructions need to be unset. This is achieved by augmenting the ROB with a 'poison' bit on the overwritten register from the rename map table. If the poison-bit is set, then the bit needs to be set in the dependency matrix.

5.4.3 Dependency Tracking After A Load Hit

When a load finally hits in the cache, dependent instructions can begin to drain from the instruction queues. Once the long-latency miss is over, newly fetched instructions which are part of a predict-miss dependency chain do not need to be dispatched to FIFOs.

We cannot simply clear the dependency chain and stop tracking dependencies on this finished load without incurring a deadlock. A deadlock condition will arise if the dependency chain does not "hand-off" dependents on the FIFO queue to the issue queue. Instructions steered to FIFO queues do not broadcast results on the writeback buses to the issue queues and cannot wakeup issue queue instructions. All recipients of data written by FIFO instructions, by necessity, *must* enter the FIFO queues. However, now that the long-latency event is over, in-order scheduling will limit ILP and performance.

We are able to transition the load-dependency chain from the FIFO queue to the issue queue by using a special dependency chain, shown in Figure 5.6, activated after the load has hit in the cache. This dependency chain continues tracking those instructions still resident in the FIFO queues but whose parent load is no longer pending. Younger instructions with a source in this IQ broadcast chain are steered to a FIFO, but configured to broadcast their writebacks to the issue queue. These younger instructions do not set a bit in any predict-miss column in the matrix, and all dependents will be steered to the "normal" issue queue CAM. This IQ broadcast chain, ensures the processor does not deadlock by allowing this subset of FIFO instructions to wakeup issue queue dependents.

When the predicted cache-miss returns with data, the load performs the column "hand-off" by

'ORing' the remaining bits of its column into the separate IQ broadcast chain. Now it can deallocate its own chain by resetting all bits to 0. Bits are cleared in this IQ broadcast column with the same mechanism as all other bits in the dependency matrix, when the logical register is re-mapped to a new physical register.

Figure 5.7c illustrates the load return. Load 0x0A returns and clears the chain. It's remaining bits are still 'live' in the FIFO, and are OR'd into the IQ wakeup vector. These bits do not get transferred if any of these bits were set in any other predict-miss chain. They are not set in this case because the dependents will still be dispatched to the FIFO and do not need the issue queue wakeup.

Instructions with a source in this IQ wakeup chain will be dispatched to the FIFO queue because they will not receive a traditional issue-queue broadcast for their sources. However, these instructions are configured to broadcast their result to the issue-queue, even though they were steered to the FIFO. Bits in this IQ chain are cleared when the register is renamed by a younger instruction, and is cleared as part of the same 'row-clear' operation. This is shown in Figure 5.8a, where register R0 is renamed. The source R0 is in the FIFO and may not have been executed yet. The target R0 will broadcast its result to the issue queue, so it clears the bit in the IQ vector.

5.4.4 Clearing a Chain

Chains are cleared in one of three ways.

1. When the load heading the dependency chain returns. The chain is cleared because the long-latency event is over and instructions do not need to slice out any more.

2. When no more instructions depend on the chain-head load. This occurs when all the bits that were set in the vector are cleared by new, younger instructions at rename or the instructions setting the bits were squashed.

3. When the load heading the dependency chain is squashed. If the head is squashed due to incorrect branch speculation, all younger instructions will be squashed.

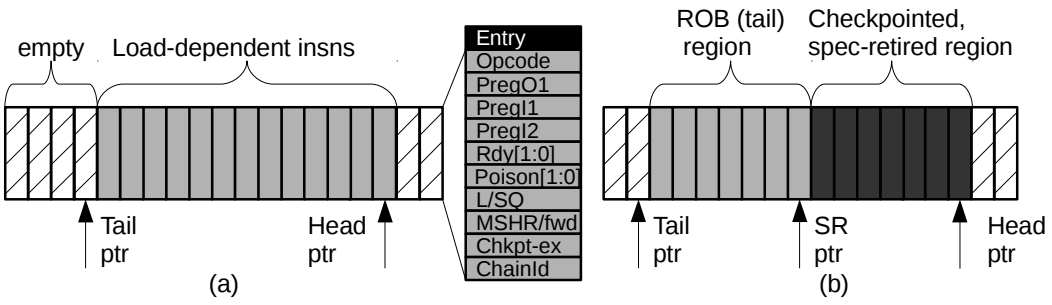


Figure 5.9: SCREW FIFO queue.

We have previously discussed the load-return case. Figure 5.8b shows the dependency chain is cleared when all dependencies are remapped. Chains can still be active, even when cleared due to register renaming. The chain is only erased when the load which created them writes back, com mitts its result or is squashed. Even if the chain is empty, meta data is still required for entering speculative retirement (described later), and therefore the chain cannot be erased from the matrix simply because all dependents have been overwritten.

5.5 In-Order FIFO Queue

The FIFO queue is responsible for scaling the scheduling window to support a large number of in-flight instructions. It is difficult to design very large, low-power issue-queue CAMs due to the number of comparators that are required to check every source against every writeback broadcast. The FIFO queue does not require comparisons against each writeback broadcast to wake-up instructions. Instead, instructions at the head of the FIFO read the register scoreboard to determine if they are ready to execute. When ready, the instructions issue from the head of the FIFO in program order.

Latency-dependent instructions are dispatched into the in-order queue FIFO, where they reside until the long-latency cache miss data is retrieved from memory. Figure 5.9a shows the FIFO queue during normal operation while Figure 5.9b shows the queue during speculative retirement mode. In both modes, younger instructions are added at the tail pointer. Each queue entry contains the same information that would be present in the issue-queue: opcode (16b), source and destination register tags (24b), and status bits(2b) for the input operands. The queue contains an additional

'checkpoint-execute' bit, indicating that the entry has executed in a checkpoint. Finally, the queue contains the 16b chain-id with the opcode of the first chain to poison the instruction. The queue also contains information that would be present in the ROB, including: load or store queue index for checkpointed loads and stores (8b), the MSHR number for loads that miss:the cache or the store-queue index for store-to-load forwarding (8b), for a total of 75 bits.

5.5.1 FIFO scheduling

During normal operation, the queue behaves shown in Figure 5.9a behaves as a simple FIFO. The head pointer indicates the front of the queue with the oldest instruction. This instruction will check the register scoreboard to determine if its source operands are ready. When ready, the instruction will acquire a functional unit and be issued, removing the entry from the head and incrementing the head pointer to examine the next instruction. These instructions are examined in priority over the issue queue, *i.e.* for the purpose of scheduling, the instruction is forced to look 'older' than any issue-queue instruction. For a four-way issue architecture, such as Nehalem, up to four instructions may issue from the FIFO in one cycle. If fewer than four instructions issue to execution units, then the remaining bandwidth can be used by the issue queue. We give priority to the queue with the oldest instruction in order to quickly free up resources from the oldest instructions in the processor. If the issue queue is given de-facto priority, then the FIFO instructions can become the oldest instruction in the ROB, preventing younger instructions from dispatching. Younger instructions may be given priority to execute, preventing this older instruction from completing and retiring until there is sufficient issue bandwidth or no structural hazards preventing the instruction from issuing.

5.5.2 Assigning FIFO to Dependency Chain

When a load is predicted to miss and a new dependency chain is assigned to it, the dependent instructions need a FIFO target to be dispatched into. Each FIFO keeps track of the total latency of all instructions dispatched but not yet issued from it. When an instruction is inserted into a FIFO, the latency accumulator is incremented with the execution latency of the instruction (e.g. add = 1 cycle). Loads are assigned a fixed latency dependent on the prediction about the cache miss

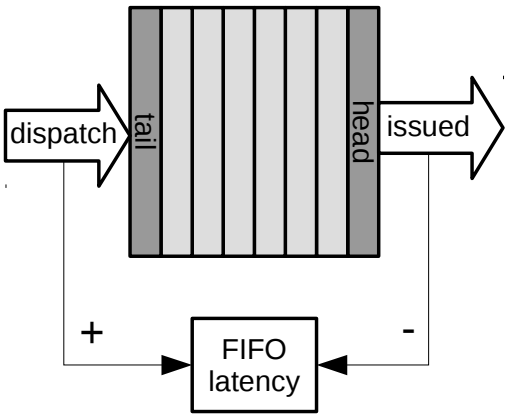


Figure 5.10: FIFO latency is incremented with each instruction dispatched to the tail of the FIFO. Latency is decremented when instructions issue.

behaviour. A predict-miss chain will dispatch all of its instructions to the FIFO with the lowest total latency when the load prediction is made. This prevents a chain's instructions from either being stuck behind other long latency instructions or not being able to dispatch because the target FIFO is full. When there is only one FIFO available, predict-miss-dependent instructions are dispatched to the same FIFO. Figure 5.10 shows how the accumulator is incremented when new instructions are dispatched to the tail, and decremented when instructions issue from the head pointer.

SCREW is also able to make use of the FIFOs in the case where the issue queue is full and would otherwise stall dispatch of instructions. When a FIFO is empty, but the issue queue is full, a predict-hit dependency chain will be assigned to a FIFO and treated as if it were a predict miss. This optimization prevents the front-end from stalling due to insufficient issue queue entries.

5.6 Speculative Retirement Regime

During a long latency cache miss, execution can be blocked due to resource starvation. A load missing the cache can cause the ROB to fill up and prevent younger, independent instructions from dispatching and executing. This same load can also cause register starvation, where instructions are unable to commit their results and free the overwritten physical register. SCREW implements speculative retirement to prevent this starvation. When the oldest instruction in the ROB is a load cache-miss, SCREW will create a checkpoint of the architectural state and enter a speculative

retirement window. SCREW will remove the load and younger instructions from the ROB and release the over-written registers of finished instructions, making the registers available to younger instructions.

When a load misses the cache, it's dependency chain is marked as 'checkpointable', and all instructions belonging to that chain are able to enter the checkpoint. Instructions which are independent of the miss are not retired until they have finished execution, *i.e.* they are not retired any earlier. A new load can miss in the cache and create a new checkpointable chain whose instructions can be added to the FIFO.

Figure 5.9b shows the FIFO queue during the speculative retirement regime. The queue FIFO is unique in that it straddles both conventional and speculative retirement regions and simultaneously contains both checkpointed (head) and ROB-managed (tail) instructions. Instructions shaded in grey are present in the ROB. These will always be the younger instructions from the tail of the FIFO to the speculative retirement pointer. Instructions shaded in dark grey are checkpointed and have been released from the ROB. Where previous latency tolerant designs include large 64-bit fields for instruction operands, SCREW is optimized for area and energy efficiency. The FIFO queue functions similar to both a ROB and instruction queue during the speculative regime.

SCREW begins speculative retirement when the head of the ROB is a load instruction that is experiencing a cache miss. When speculative retirement occurs, checkpointable instructions in the FIFO queue can (with two exceptions) immediately release their ROB entry when they are the head of the ROB. The first exception are conditional branches. It is not desirable to add too much speculative-*execution* into the checkpoint, as a branch flush will flush back to the pre-checkpoint state, which could be many hundreds of instructions earlier.

The second exception affects store instructions. Store instructions can block speculative retirement if the data register depends on a load miss. Store instructions *must* calculate their effective address and update the store queue entry prior to retirement, but they will not issue until both operands are ready. This can cause problems in speculative retirement because loads will need to check the store bloom filter to check for any store-to-load memory dependencies. If the older store

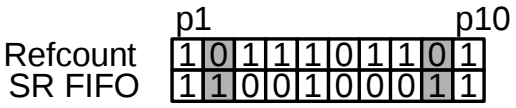


Figure 5.11: Extra reference count bitvector for managing speculative retirement in SCREW. Instructions in the FIFO older than the speculative retirement pointer set a bit in this new row to pin their registers.

has not issued, a younger load will not see that it has a dependency through memory. Memory consistency must be maintained within the speculative-retirement region, and a load executing ahead of a store will need to re-execute after the store has finished.

During speculative retirement, a speculative retirement-pointer (SR-ptr in Figure 5.9(b) is maintained separately from the head-pointer. The SR-pointer is incremented each time a FIFO instruction is added to the checkpoint. If speculative retirement is blocked with a store at the head of the ROB, the store is 'agen'd', where it is issued with the sole purpose of updating the store queue with the destination address. After the agen is complete, the store can be added to the checkpoint, and younger loads will capture the memory dependency if present. Store instructions will block speculative retirement if the address register is 'poisoned', *i.e.* if the address depends on a load cache miss. This is a less frequent case.

5.7 Physical Register File

BOLT's scaling of the physical register file relies on the fact that it releases the overwritten registers of *all* (miss-dependent or miss-independent) instructions at speculative retirement. It then re-renames miss-dependent instructions to re-execute them. Such a design is not appealing to SCREW for a couple reasons. First, BOLT's register reclamation scheme *requires* miss-dependent instructions to "capture" their miss-independent inputs. SCREW aims to avoid these value copies. Second, re-renaming instructions itself has a non-trivial energy cost.

Instead of releasing the overwritten registers for all instructions, SCREW only releases the overwritten registers which are independent of the miss. This approach allows the (miss-dependent)

instructions in the FIFOs and issue queue to execute naturally, with no need to re-rename them, as both their input and output registers remain valid.

The trick here, is *how* to manage the registers in this fashion (in an efficient way). To implement this scheme, SCREW adds rows to its reference count matrix [73], which represents registers used by instructions in the FIFO and issue queue older than the speculative retirement pointer. These rows allow SCREW to force these miss-dependent registers to remain live, while reclaiming all miss-independent registers.

Speculative retirement in SCREW starts in much the same fashion as in BOLT. When the head of the ROB is a load experiencing an LLC miss, SCREW takes a retirement map table checkpoint (including the associated reference count manipulations). It then begins speculatively retiring instructions from the ROB. Logically, there is a speculative retirement pointer which advances through the ROB, examining each instruction. The instructions chain-mask identifies whether or not the instruction depends on the outstanding miss. For a miss-independent instruction which is completed, the instruction is speculatively retired, releasing its reference counts on its overwritten register (as it would normally at retirement). Note that unlike a traditional ROB-architecture, releasing the reference count does not guarantee that the register is freed. The register will only be freed if nothing else holds a reference count for it. For a miss-independent instruction that is not completed, speculative retirement waits until the instruction is completed.

For a miss-dependent instruction, the speculative retirement stage examines the physical registers that it uses (reads or writes), and sets their reference count bits in the newly added row of the matrix. If these bits are already set (*i.e.*, an older instruction in the FIFO reads and/or writes the same register), that is fine. The bit remains set, and, as we will see shortly, an exact count is not required, only a set bit.

Because the FIFO's row in the reference count matrix only contains a single bit per register, SCREW cannot clear them on an instruction-by-instruction basis. Instead, it must clear the entire row completely whenever the actual retirement pointer catches up to the speculative retirement pointer (meaning that there are no instructions in the FIFO older than the speculative retirement

pointer). Such a situation can occur after a load miss returns, and the FIFO instructions execute, or after a mis-speculation squashes speculatively retired instructions, restoring a previous checkpoint.

SCREW employs a hybrid register management scheme, similar to BOLT [35], but with a few key differences. As mentioned earlier, SCREW's execution is split into two regimes or instruction windows: the conventional ROB-window and the checkpointed/speculative retirement window. Conventional ROB and free-list register management is the default regime. Instructions proceed through the ROB unless a load misses the cache and blocks the head of the ROB from retiring. In this mode, the ROB manages instructions and provides instruction-level recovery. This is important, because speculative instructions—*i.e.* instructions fetched beyond a branch—will not be checkpointed and added to the speculative retirement region unless the processor is resource limited and the branch must be checkpointed to prevent further stalls. The reason branches are not checkpointed more aggressively is because a branch mispeculation will squash the entire checkpoint, which can contain many hundreds of older instructions. In a ROB-managed region, a mispeculated branch will only squash *younger* instructions. The ROB-managed instruction window will therefore contain many more speculative instructions.

BOLT's scaling of the physical register file results from releasing overwritten registers of *all* (miss-dependent or miss-independent) instructions at speculative retirement. BOLT then re-renames miss-dependent instructions to give them a new destination register and re-execute them. Such a design is not appealing to SCREW for a couple of reasons. First, BOLT's register reclamation scheme *requires* miss-dependent instructions to access the register file and "capture" their miss-independent inputs. SCREW aims to avoid these expensive value copies. The second portion, re-renaming, also has a non-trivial energy cost to access the freelist and rename map tables. Each dependent instruction must be re-renamed and update its source pointers to get the new mapping.

Instead of releasing the overwritten registers for all instructions, SCREW only releases the overwritten registers for miss-independent instructions. These registers are not required to execute instructions in the FIFO. This approach allows the miss-dependent instructions in the FIFO to execute without needing to be re-renamed, as their input and output register mappings have remained

valid.

To implement this scheme, SCREW adds a row to its reference count matrix [73, 4], which represents registers used by instructions in the FIFO older than the speculative retirement pointer. This row pins the registers, forcing them to remain live while allowing SCREW to reclaim all miss-independent (non-FIFO) registers. Figure 5.11 shows this, highlighting two cases in grey where registers would be released if they were not pinned. The FIFO instructions have executed but are older than the checkpoint and must wait for the checkpoint to be released.

Speculative retirement in SCREW starts in much the same fashion as in BOLT. When the head of the ROB is a long latency load, SCREW takes a retirement map table checkpoint (including the associated reference count manipulations to identify registers as belonging to the retirement map checkpoint). It then begins speculatively retiring instructions from the ROB. Logically, there is a speculative retirement pointer which advances through the ROB, examining each instruction.

For a miss-independent instruction which is completed, the instruction is speculatively retired, releasing its reference counts on its overwritten register (as it would normally at retirement). Note that unlike a traditional ROB-architecture, releasing the reference count does not guarantee that the register is freed. The register will only be freed if nothing else holds a reference count for it. For a miss-independent instruction that is not completed, speculative retirement waits until the instruction is completed.

For a miss-dependent instruction (in the FIFO), the speculative retirement stage examines the physical registers that it uses (reads or writes), and sets their reference count bits in the newly added row of the matrix. If these bits are already set (*i.e.*, an older instruction in the FIFO reads and/or writes the same register), the bit remains set.

Because the FIFO's row in the reference count matrix only contains a single bit per register, SCREW cannot clear them on an instruction-by-instruction basis. Instead, it must clear the entire row completely whenever the actual retirement pointer catches up to the speculative retirement pointer (meaning that there are no instructions in the FIFO older than the speculative retirement pointer). Such a situation can occur after a load miss returns, and the FIFO instructions execute, or

after a mis-speculation squashes speculatively retired instructions, restoring a previous checkpoint.

5.7.1 Recovery

As with any micro-architectural design, SCREW must not only account for "normal" operation, but also include techniques to recover from mis-speculations. Many aspects of recovery are similar to BOLT. Mis-speculations of speculatively retired instructions can be recovered by restoring an older retirement map table checkpoint. Also, as with BOLT, the instruction granularity commit expectations of SVW are reconciled with checkpoint based register commit/recovery via DSC/SDR [34].

However, SCREW must also deal with recovering the dependency matrix. When the pipeline is flushed, the dependency matrix must be recovered along with the rename map table and reference counts. The rename map table itself is generally recovered in one of two ways, either by restoring a previously taken checkpoint (*e.g.*, at a low confidence branch), or serially by traversing the ROB backwards and "un-renaming" each instruction.

Rather than checkpoint the dependency matrix and track which chains are still active, we rebuild the matrix after flush. Any pipeline flush will drain the pipe of instructions younger than the flush, starting from the tail of the FIFO up to the head. When a load is flushed, its column in the dependency matrix is cleared and all instructions in the FIFO younger than the load will be squashed. Squashed instructions also clear the row associated with their logical destination register. This prevents instructions fetched from the correct path from being dispatched to a FIFO due to dependencies set by wrong-path instructions.

A similar process needs to trim the remaining dependency chains, to ensure that instructions in the IQ can wake up. We walk the FIFO from the new tail pointer up to the head pointer, setting the destination register bit in the IQ broadcast chain for all instructions in the chain, and 'ORing' any remaining bits from the remaining columns. This ends the pre-flush dependency chains prematurely, but ensures that dependency chains are closed around the flush.

5.8 Load Queue/Store Queue

For the most part, SCREW can make use of the scalable load queue and store queue designs used in BOLT. The only difficulty here arises in stores which have a poisoned data input, but not a poisoned address input. In BOLT, such stores compute the physical address they will write when they pseudo-execute. Knowing their target address is important, as it allows the store to write the root table of the chained store buffer during speculative retirement. BOLT must stall speculative retirement on stores whose address input is poisoned, however, BOLT (and prior LT work) show that such stores are quite rare.

Stores with poisoned data inputs are too prevalent for SCREW to stall on them. However, SCREW does not perform pseudo-execution in the same way BOLT does. If the underlying micro-architecture splits stores into address μops and data μops, then the address μop would execute normally, and everything would work in a straight forward fashion. However, if this splitting is not done in the general case, SCREW has to perform an "address execution" for stores which would otherwise block speculative retirement.

Chapter 6: SCREW Performance Evaluation

This chapter presents an evaluation of the SCREW microarchitecture, comparing performance and efficiency against two microarchitectures: a recently proposed latency tolerant microarchitecture (BOLT) [35] and a conventional re-order buffer (ROB)-based out-of-order processor microarchitecture [8]. We are interested in how latency tolerance can be applied to small out-of-order processors, where area and energy are both first order constraints. The baseline processor models a 'small out-of-order' 2-way superscalar AMD Bobcat processor [8]. This is in contrast to a 'large' out-of-order processor, such as a 4-way superscalar Intel Core i7 processor [95]. The BOLT processor applies BOLT-style latency tolerance to this Bobcat baseline.

Each simulated microarchitecture attempts to faithfully represent the Bobcat processor using publicly available information to model each aspect, including queue depths, structure sizes, and issue widths. There are two exceptions where we do not precisely follow the baseline microarchitecture: the first is the branch predictor where we use a three-table PPM [48] predictor, rather than a two-level predictor as in Bobcat. The second exception is that all configurations make use of SVW [72] and SQIP [78] load/store queues. This removes the load/store queue as a variable, allowing us to making more direct energy and performance comparisons between the latency tolerant and baseline architectures. Table 6.1 lists the salient features of each processor.

We target energy efficient single-threaded performance, and evaluate each processor microarchitecture using the single-threaded SPEC2006 benchmarks listed in Table 6.2. We use a cycle accurate x86 processor simulator to model the architectures and their features. Out simulator periodically samples the workload, fast-forwarding for 480 Million instructions before a 10 Million instruction cache and branch predictor warm-up phase. Following this phase, a 10 Million instruction detailed simulation is performed. We simulate for ten of these epochs, yielding a detailed sampling of 100 Million instructions across 5 Billion instructions in the workload.

We are particularly interested in memory bound benchmarks, *i.e.* those benchmarks which

exhibit low IPC and moderate levels of MLP highlighted in **bold**. Memory-bound workloads can experience significant delays in the typical out-of-order microarchitectures because while the pipeline can schedule around an L1 cache miss, the out-of-order pipeline cannot tolerate the latency for accesses to main memory. The queueing structures and register files would need to grow too large to absorb the latency to main memory. Contemporary processors have attempted to solve this problem by supporting wider levels of multithreading. When one thread stalls due to a cache-miss, another thread can still execute. However, limits of thread-level parallelism [32] require solutions which can address single threaded performance.

6.1 Energy Model

Our energy model includes both static and dynamic components. We record activity of critical pipeline structures along with static energy from CACTI [37] to compare microarchitectures. This model accounts for the static costs of new structures and accesses to pipeline-critical structures in each microarchitecture, along with the costs for steering and executing instructions. We compare relative cycle counts, micro-instruction execution counts, register file accesses and rename activity, slice buffer accesses, issue queue broadcasts, and predictor accesses between the baseline micro-architecture and BOLT and SCREW latency tolerant microarchitectures.

$$\frac{E_{LT:dyn}}{E_{base:dyn}} = \alpha \times \frac{E_{LT:RF}}{E_{base:RF}} + \beta \times E_{LT:slice} + \gamma \times \frac{E_{LT:iq}}{E_{base:iq}} + \delta \times \frac{E_{LT:rem}}{E_{base:rem}} \tag{6.1}$$

6.1.1 Dynamic Energy

Equation 6.1 describes the dynamic energy for the processor. This includes the cost to access the register file, the cost to slice out instructions, and the remaining energy to execute and complete the instruction. This equation models accesses to the register file. It is important to model the register file carefully, as it can consume up to 30% of dynamic power [17, 96]. We account for this fraction of core power with α in equation 6.1. Each latency tolerant scheme will have more accesses to the register file, as they typically execute more instructions in the larger scheduling window. The BOLT micro-architecture will have additional register reads as values are copied from the register-file

Table 6.1: Simulated processor configurations.

2-way Bobcat			
Item	Baseline OoO	BOLT	SCREW
Clock	1600 MHz		
BPred	3-table PPM: 256×2, 128×4, 128×4 8b tags, 2b counters		
IQ	24 entries		
Regfile	120 pregs (64 arch + 56 rename)		
ROB	56 entries		
L1 I$	32KB, 256-set, 2-way, 64B blocks, 3-cycles		
L1 D$	32KB, 64-set, 8-way, 64B blocks, 3-cycles		
L2 $	512KB, 512-set, 16-way, 64B blocks, 16-cycles		
Mem	tCAS: 10ns, tRAS:45ns, tRP:15ns[1]		
Slice Buffer	—	64	64
Chkpoints	—	2	2
SQ type	SQIP	SQIP/CSB	SQIP/CSB
SQ size	22	64	64
LQ type	SVW	SVW	SVW
LQ size	24	96	96

[1]Memory latency is set for a 90ns average latency per [3]

Table 6.2: SPEC2006 Benchmarks

Benchmark	Input	IPC	L2 MLP	L2 MPKI	BP Acc.	Avg. Regs	Avg. Inflight
cactusADM	test	0.62	4.3	2.0	98.6	102.4	56.6
calculix	train	1.15	1.1	0.9	96.6	85.6	39.9
dealII	test	0.88	1.3	1.9	97.6	93.6	51.5
gamess	test	1.03	1.3	0.5	96.7	96.6	50.1
GemsFDTD	test	1.15	2.0	4.5	97.7	78.1	32.5
lbm	test	0.36	6.2	40.3	98.7	97.2	53.1
milc	test	0.31	1.9	17.3	97.3	108.2	58.8
namd	train	0.99	1.4	0.1	97.6	97.1	49.0
povray	train	0.96	1.6	0.0	95.2	88.2	43.9
soplex	train	0.33	1.7	21.2	95.5	99.5	55.3
sphinx3	train	0.81	1.9	0.6	97.1	95.4	50.8
tonto	test	1.10	1.4	0.1	97.1	87.7	43.6
wrf	test	0.94	1.6	3.9	98.8	100.4	53.9
zeusmp	test	0.63	1.5	4.2	97.4	106.5	60.3
astar	test	0.67	1.5	0.5	78.9	78.4	32.3
bzip2	train	0.93	2.1	2.5	95.5	88.3	46.7
gcc	train	0.74	1.3	1.7	95.7	82.0	41.6
gobmk	train	0.72	1.3	1.2	86.3	75.5	27.4
h264ref	test	1.22	1.4	0.9	96.4	87.2	42.7
hmmer	test	1.15	1.4	0.0	92.4	81.2	34.2
libquantum	train	1.64	1.4	0.4	98.3	84.1	37.5
mcf)	train	0.14	2.3	48.0	89.6	87.9	52.9
omnetpp	test	0.88	1.2	0.1	90.9	72.7	24.0
sjeng	test	0.70	2.5	11.0	88.5	72.2	26.2
xalancbmk	test	0.75	1.5	0.2	91.1	74.2	26.3

Table 6.3: Area increase vs. $4.6mm^2$ Bobcat Core

SCREW	Area	BOLT	Area
Load predictor	0.0130	ROB operand	0.0200
FIFO queues	0.0186	Slice buffer	0.0332
Dep. matrix	0.0046	Vectors	0.0011
Bypass table	0.0014	Ptr chase table	0.0014
Reference Counts	0.0011	Reference Counts	0.0011
Total	0.0387	Total	0.0568
Overhead	0.85%	Overhead	1.23%

into the slice-buffer during the slice-out phase, incurring a register-read and slice-buffer write. We account this energy in both the register file and slice buffer components of our model.

The baseline OoO configuration does not have a slicing component, so $\beta_{OoO} = 0$. In BOLT and SCREW, the slice energy contains the cost to read and write to the slicebuffer or FIFO queue. The slicebuffer and FIFO queue are built as multi-ported RAM, and are modeled similarly to the register file. To estimate the overhead, we count the number of accesses to the buffer and scale by α and the relative energy cost between a slice buffer and register file access.

For SCREW, we also include the cost of the load predictor. We model the load predictor cost as a branch predictor, scaling by the relative number of accesses and updates. Branch predictors, especially those with multiple history tables, can contribute 7% to 10% of core power [59, 58]. We conservatively choose 7% for our low-power out-of-order core.

The issue-queue is the final explicitly modeled component of our dynamic power model. The issue queue can contribute up to 25% of core dynamic power, with 90% of this due to operand wakeup broadcasts [52]. These broadcasts are expensive because every instruction in the issue queue must compare its source register tags with each of the four register tags that can be written per cycle. SCREW eliminates broadcasts for sliced instructions as the dependents are captured in the FIFO queue and are issued in FIFO order.

6.1.2 Static Energy

We approximate static energy with Equation 6.2 which is a function of leakage current of the core and the execution time of the workload. We assume that the core is power gated once the workload is complete to prevent idle leakage. We use an area-proportional model of static energy, where we assume each new component will leak at the same rate as the rest of the core. Table 6.3 shows the area overhead of each new structure in BOLT and SCREW which corresponds to a proportional leakage increase. We assume that this static energy component corresponds to 25% of core power in our model, approximating deep sub-micron technology similar to Power 7 [96].

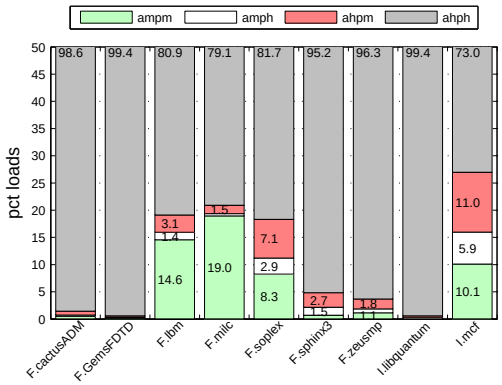
$$\frac{E_{LT:sta}}{E_{base:sts}} = \frac{cycles_{LT}}{cycles_{base}} \times \frac{(i_{sta:LT:overhead} + i_{sta:base})}{i_{sta:base}} \tag{6.2}$$

6.2 Load Behaviour Prediction

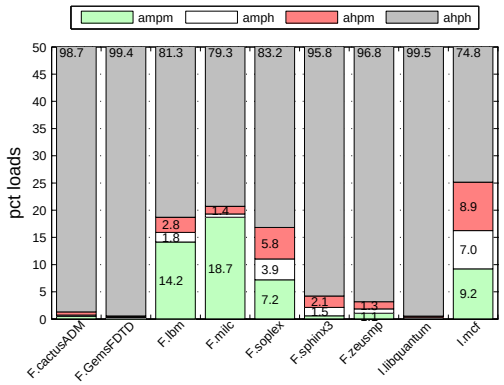
We first evaluate the load-miss predictor in the SCREW pipeline to measure how well we can predict last level cache access behavior using only the instruction address and prior history. The metrics of merit include hit/miss coverage—the number of correctly predicted hits/misses as a fraction of all hits/misses—and hit/miss accuracy—the number of correctly predicted hits/misses as a fraction of hit/miss predictions made. Higher coverage and accuracy is the goal. However there is a trade-off; higher miss-coverage will predict more misses, but can result in lower miss-accuracy, as more hits are incorrectly predicted as misses. Predictions are made at the front-end of the pipeline at instruction rename and are updated when the instruction retires.

The first predictor type is a two-stage local predictor. This predictor records the history of the previous 11 accesses to a single counter. The history is selected via the upper 10-bits of the load instruction address, while the 11-bits of history index into a table of 2-bit counters. This predictor requires 10Kb for the pattern history table and 4Kb for the counter table, for a total of 15Kb of state. We have two versions of this predictor—the first predicts a miss if the counter is greater than 0, while the second is less 'aggressive' and predicts a miss if the counter is greater than 1.

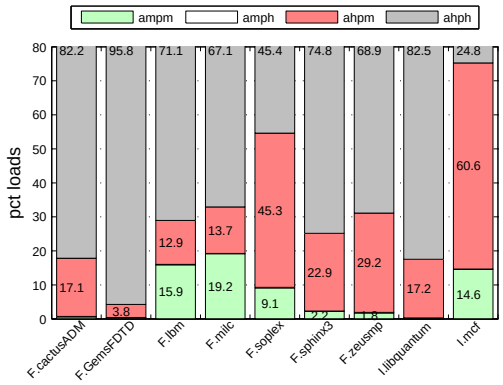
The next predictor is a 'hybrid' predictor modeled after the Alpha21264 [41] predictor. This



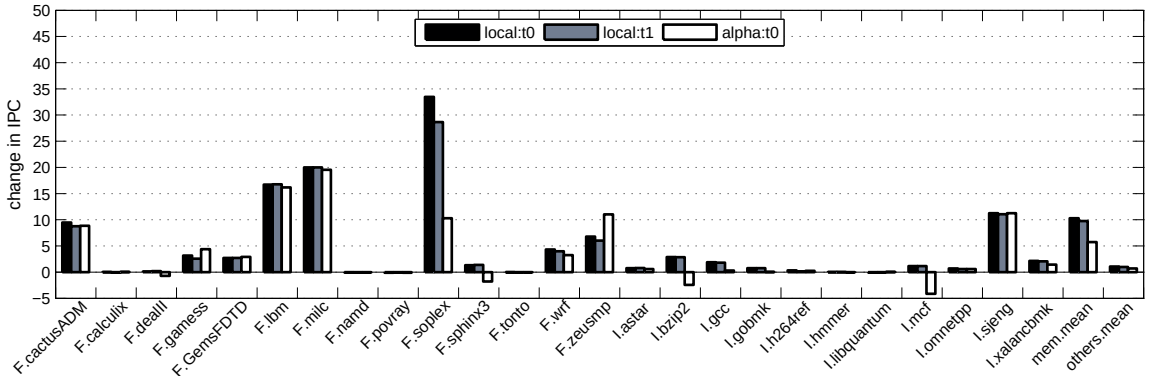
(a) Local t0



(b) Local t1



(c) Hybrid (Alpha-21264)



(d) Change in IPC

Figure 6.1: Local and Hybrid Predictor performance. (a)—(c) classification of loads for each predictor and observed behavior. (d) IPC for each predictor configuration. Harmonic mean IPC of memory-bound and 'other' non-memory-bound workloads are shown on right.

predictor is comprised of 2 component predictors (local and gshare), with a selector to pick the 'winning' component. This predictor is designed to reduce aliasing among instructions with similar addresses or behaviors. Some loads may correlate well to the global history of all loads, while other loads will correlate better with their own previous history.

Unlike cache predictors which have the benefit of only seeing accesses that have already missed one or more levels of the cache hierarchy, there is no 'inherent' filtering applied to the predictor, each load fetched and decoded will access the predictor. The majority of load instructions will hit in the L1 cache due to the nature of spatial and temporal locality. With such behavior, pattern history tables will often be identical for many load instructions leading to aliasing for many loads.

Figure 6.1(a)—(c) shows the fraction of correct and incorrect L2 cache hit- and miss-predictions for the local 0-threshold (a), local 1-threshold (b) and Alpha predictor (c). Each column is a stacked bar plot where each layer in the stack classifies the load prediction behaviour as one of the four classifications: (i) correctly predicted load misses—actual-miss, predict-miss (ampm); (ii) incorrectly predicted hits—actual-miss, predict-hit (amph); (iii) incorrectly predicted misses—actual-hit, predict-miss (ahpm); and (iv) correctly predicted hits—actual-hit, predict-hit (ahph). The ampm bar at the bottom represents the fraction of loads that were correctly predicted to miss. These loads form the heads of dependency chains which are sliced to the FIFO queue. Above this bar is the fraction of load misses which were incorrectly predicted to hit. This represents the missing coverage for the predictor. These loads will behave similarly to those in the baseline processor, and will fill the issue queue with waiting instructions and potentially limiting performance gains. The third bar from the bottom represents incorrectly predicted misses. These 'ahpm' loads will have their dependents dispatched to the FIFO, causing unnecessary serialization.

Increasing the counter threshold that yields a predict-miss causes a reduction in false-positives. This is most evident when comparing soplex and mcf in Figures 6.1(a) and (b). I.mcf in (b) has fewer than half as 'many actual-hit, predict misses' as (a). However, this comes at a cost of miss coverage, which reduces from (a) to (b). The predictor in Figure 6.1(c) is much 'greedier' the local predictors, and is much more likely to predict a load miss. This predictor has much higher 'miss-coverage', but

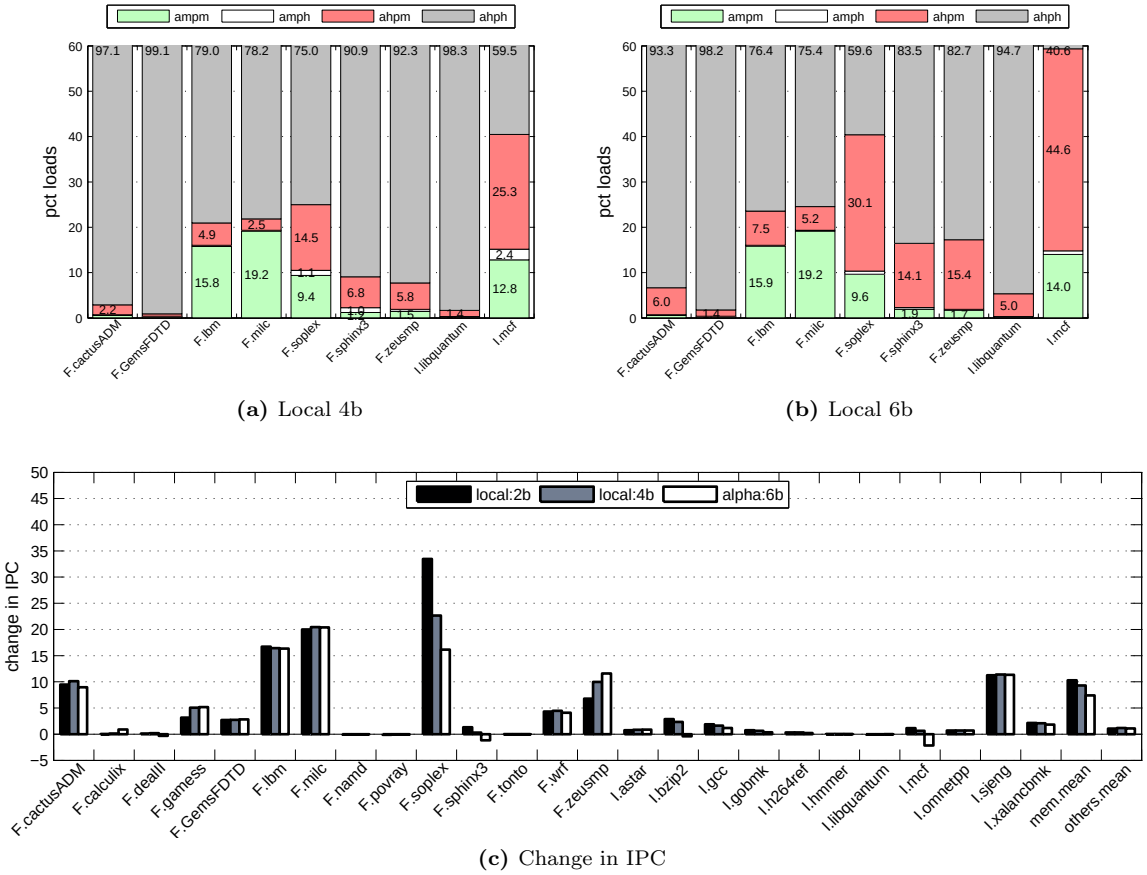
it also has many more false positives.

Figure 6.1(d) shows the effect on IPC for these three predictor configurations. Three benchmarks are particularly sensitive to the predictor configuration. Performance in F.soplex and I.mcf drops significantly from the local predictor to the 'greedier' Alpha predictor. The Alpha predictor has many more false-positives, filling the in-order FIFOs and stalling the front end of the pipe. These instructions stuck in the FIFO have a longer 'lifetime' from dispatch to retirement, and keep their source registers alive longer, since the data is not read until after the instruction is issued. If the FIFOs could schedule out-of-order, this wouldn't be a problem. This is not the case in these FIFOs. An 'actual-hit, predict-miss' load will issue from the issue queue and hit in the L1 cache, but its dependents are stuck in the FIFO behind instructions waiting for an L2 miss return. This will effectively turn the L1 hit into an L2 miss as far as the dependent instructions are concerned. F.zeusmp has the opposite behavior. This workload performs better with higher 'miss-coverage', even though it contains many more incorrect predict-miss dependency chains. The Alpha predictor steers more instructions to the FIFOs, freeing space in the issue queue. The serialization of load-hits is less of a factor here, as issue queue occupancy is the main performance limiter. A more aggressive predictor will collect more false positives, for that reason, we chose the less aggressive 'local' predictor, for the rest of our studies.

Scaling Counter Width

In Figure 6.2, we scale the size of the counters in the local predictor from 2b to 4b and 6b. A larger counter will increase hysteresis, requiring more 'hits' after a miss before the prediction flips from miss to hit. This will increase the miss coverage, as load misses are predicted more frequently.

Figure 6.2(a) and (b) shows the classification of each load. When comparing to Figure 6.1(a), miss coverage (ampm loads) increases with larger counters. However, just as with the Alpha predictor, increased coverage brings increased mis-predictions. Figure 6.2(c) illustrates the performance impact of these wider counters. More hysteresis and increased load hits steered to the FIFOs decreases performance for the same reasons as with the Alpha predictor. We use 2-bit counters for the rest of this evaluation as it yields the best performance.

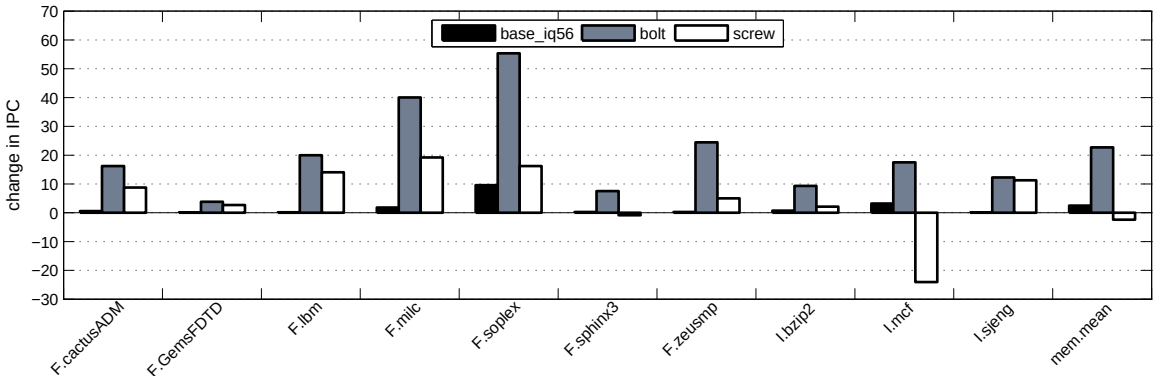


(a) Local 4b

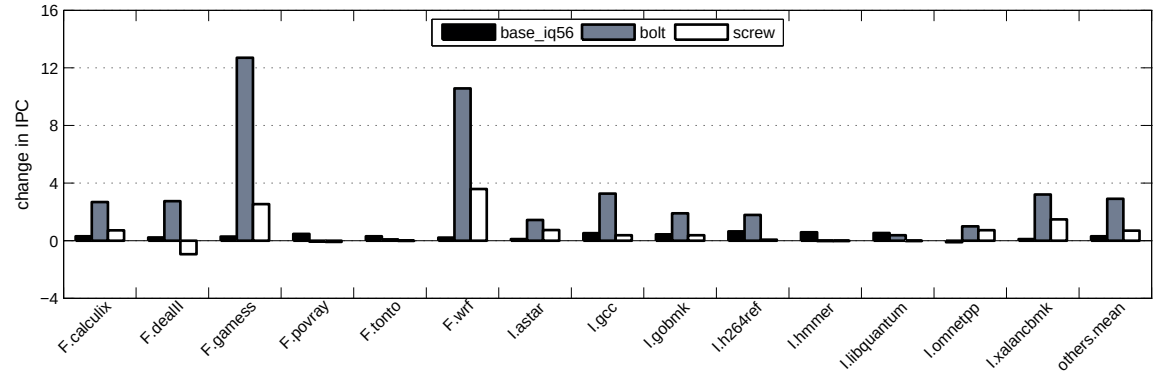
(b) Local 6b

(c) Change in IPC

Figure 6.2: Local Predictor performance when increasing counter size from 2b to 6b. (a)—(b) load classification, (c) IPC change from baseline processor. Harmonic mean IPC of memory-bound and 'other' non-memory-bound workloads are shown on right.



(a) Change in IPC (memory intensive benchmarks)



(b) Change in IPC (other benchmarks)

Figure 6.3: Performance improvement for scaled issue queue, BOLT, and SCREW processor. Change in harmonic mean IPC is shown in 'mean' entry on right.

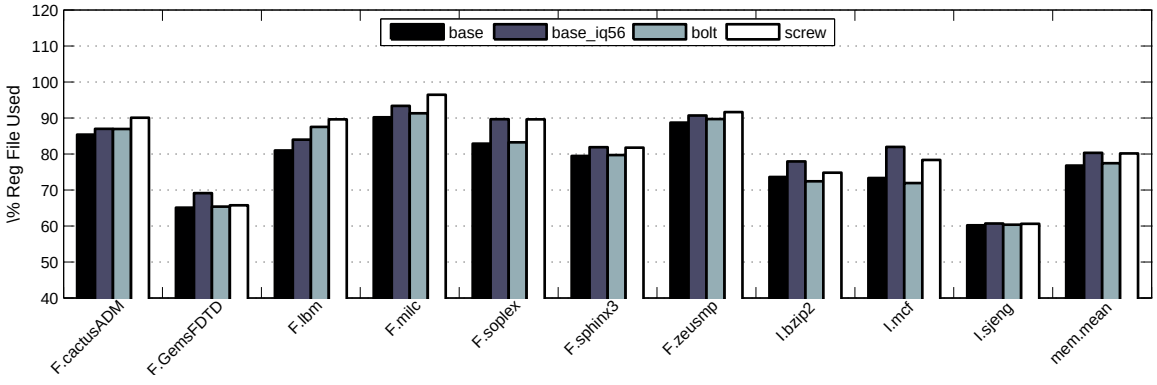


Figure 6.4: Percent register file occupied for baseline, scaled issue queue, BOLT, and SCREW processors. Larger instruction windows will typically require more registers.

The issue queue in BOLT is the same 24-entry queue as in SCREW and the baseline.

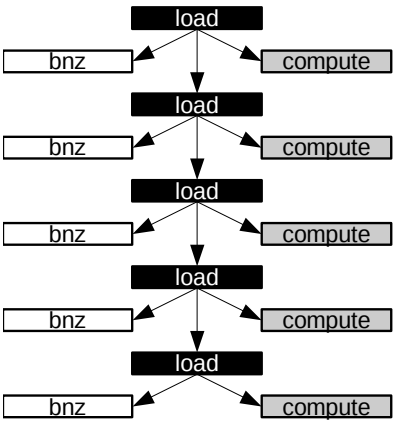
Figure 6.3 shows the change in performance *(IPC)* from the baseline for memory bound workloads (a), and all other SPEC workloads (b). The leftmost bar shows that scaling the issue queue alone is not enough to improve performance across a variety of workloads. The IQ-scaled baseline achieves a maximum 9.6% IPC improvement for F.soplex, with an average performance improvement of 2.5% for memory bound workloads. Scaling the issue queue without scaling the other resources in the pipeline will not improve performance significantly because the other window constraints will still be present. Reorder buffer and register file utilization will still limit performance once the issue queue dispatch stalls are removed.

The BOLT processor has the highest performance, with an average 23.4% improvement for memory-bound workloads. Figure 6.4 shows that BOLT has the lowest average register file occupancy of all configurations. This is a result of BOLT copying data from registers as it performs its 'slice-out' mechanism. Once the last reader has read from a register and sliced out, the register can be freed. SCREW on the other hand achieves a maximum 19.6% IPC improvement for F.milc yet has an average performance loss for memory bound workloads. In this single-FIFO configuration, average performance is dominated by a loss in the I.mcf workload, where SCREW loses 25.6% performance.

Serialization Losses

Loads frequently miss the cache in the I.mcf workload, filling the scheduling queues with stalled instructions and yielding low IPC. The baseline processor spends approximately 72% of the runtime stalled at dispatch due to the issue queue being full. For workloads like I.mcf, the issue queue will typically be full of instructions dependent on a load miss. I.mcf is interesting in that the main loop which dominates performance involves a lot of pointer chasing, where a each load feeds a younger load. Each load in the chain of loads can miss in the L2 cache, causing significant delay. Figure 6.5 shows an example dataflow for this pointer chasing loop. There is little MLP to be extracted from within a single load-chain. However, multiple independent load chains can exist in parallel. These can be scheduled out-of-order from an issue queue, but not from the FIFO in-order queue.

The SCREW processor correctly predicts that loads will miss the cache, steering 32% of instruc-




```
mcfutil.c (83:95)
while( node ) {
     if( node->orientation == UP )
         node->potential = node->basic_arc->cost +
                           node->pred->potential;
     else { /* == DOWN */
         node->potential = node->pred->potential -
                           node->basic_arc->cost;
         checksum++;
     }
     tmp = node;
     node = node->child;
 }
```

Figure 6.5: Pointer chasing. left: data-flow graph representing C-code on right, from `429.mcf` SPEC2006 benchmark. Code iterates through a linked list of nodes. Dependencies exist between `node` and `node->child`.

tions to the FIFO queue. The average occupancy is 25 entries, which is roughly equivalent to the size of the issue queue. Once in the FIFO, loads will be serialized behind other instructions which are waiting for data to return from the L2 cache. Performance would not degrade if these dependent instructions could be scheduled out-of-order, as is evident in the scaled-IQ configuration. One way to address this performance loss it to increase he number of FIFO queues available as dispatch and scheduling targets. This will spread independent load chains among multiple FIFOs, reducing the occasions where a load and its dependents are stuck behind older instructions waiting for data.

In F.soplex, SCREW reduces the delays at the front-end of the pipe, as the FIFO buffer provides a new dispatch destination. However, once dispatched to the single FIFO, scheduling delays reduce processor performance. The 32-entry FIFO queue also fills up, stalling dispatch. This does not occur in BOLT, which 'slices' instructions by removing them from the issue queue and buffering in a slicebuffer until the load returns. The SCREW FIFO queue contains both 'active' and 'buffered' instructions. A wider pool of FIFOs will alleviate the scheduling delays, while FIFOs with more entries will prevent the FIFO from limiting dispatch

As noted earlier, performance for F.zeus is limited by the predictor coverage. The predictors are not able to adequately learn the cache behavior and steer enough instructions to the FIFO queue. A

greedy predictor will improve performance for this particular workload at the cost of performance for other workloads, as hit-dependent instructions are stuck behind older miss-dependent instructions in the FIFO.

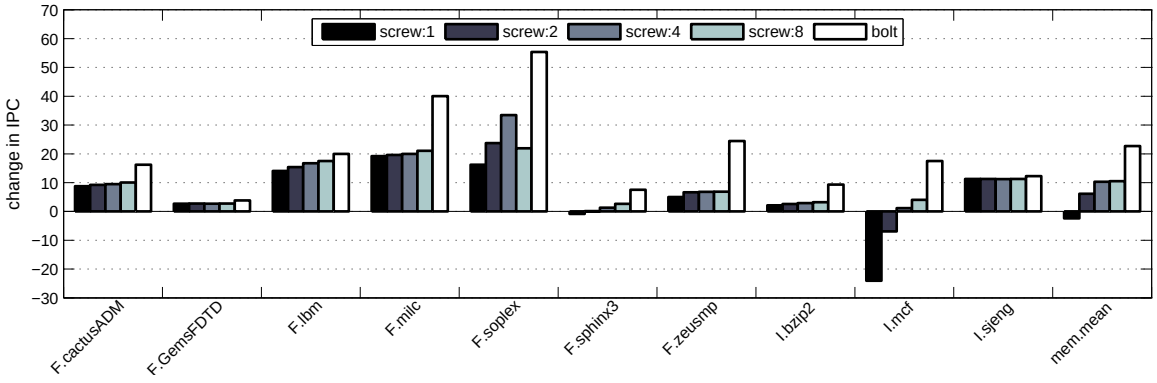
6.4 Sensitivity Analysis

In this section we evaluate how each structure in the SCREW processor affects performance. We scale the number of FIFO queues, the size of the dependency matrix, and the size of each FIFO. We expect that performance should grow with more resources, and unlock some of the performance held back by the serialization of the FIFO queues. However, there is a limit to performance gains as each cache-miss dependent instruction will occupy a register. We measure sensitivity to register file size separately. We expect the SCREW processor to be particularly sensitive to issue-queue size and register file size. The ROB is less of a constraint because SCREW can create checkpoints when a load stall blocks the head of the ROB. These checkpoints will help to release registers after instructions complete the writeback stage, but registers for un-executed instructions are still consumed. These registers can be held for a long time, especially if the instruction is dependent on a load that misses and the load was not predicted to miss the cache.

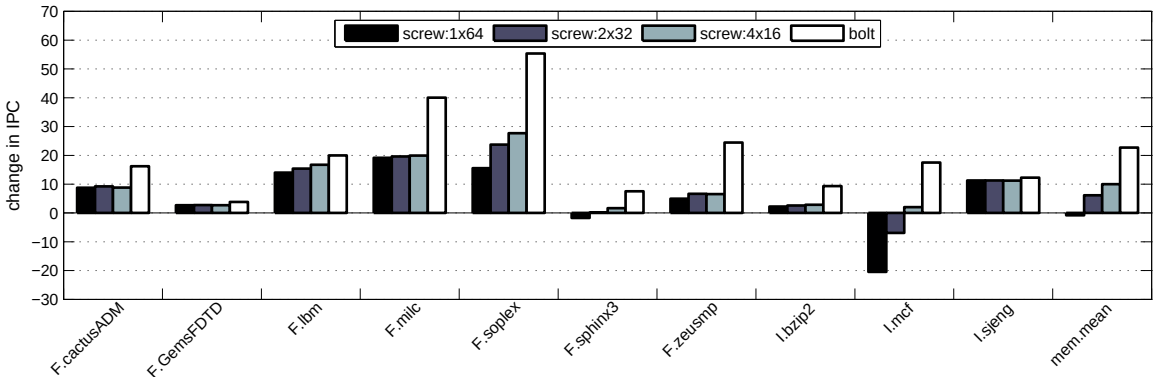
6.4.1 Scaling In Order FIFO Queues

We address the serialization losses by scaling the number of FIFO queues in the processor, as described in Section 5.5. Each load in the instruction stream is assigned a dependency chain which contains the destination registers for instructions dependent on the load (or dependent on dependents.) If the load is predicted to miss the cache, the chain is assigned to the FIFO queue with the lowest latency. This helps ensure that FIFOs contain independent load chains . A SCREW processor with more queues will support more ILP, as there will be more instructions available to the the scheduler.

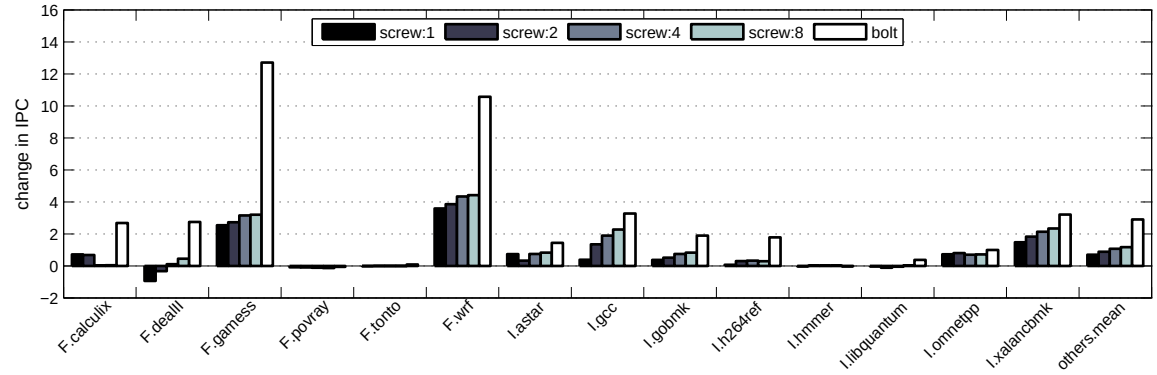
Figure 6.6 shows the performance when increasing the number of FIFOs in the system from one to eight-FIFOs. Each FIFO is 32-entries deep, with a dependency matrix capable of tracking 32 inflight loads. Figure 6.6(a) shows that the increased ILP available from multiple FIFOs reverses



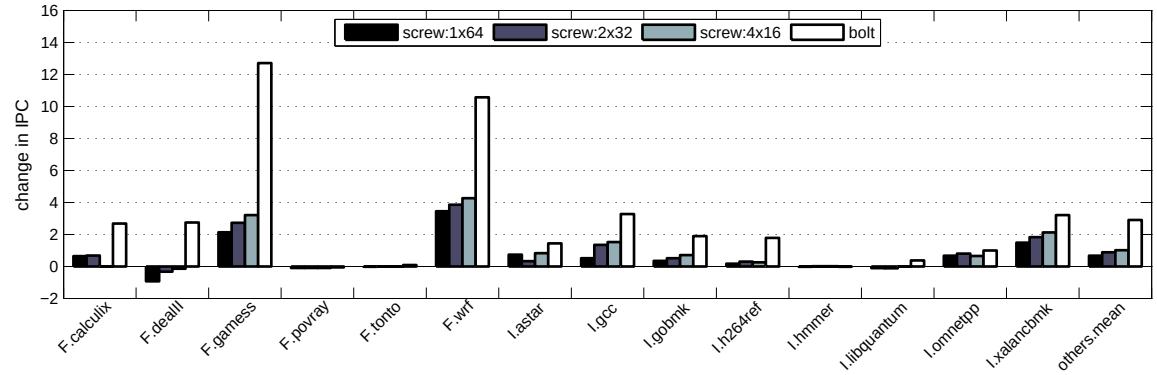
(a) Change in IPC (memory intensive benchmarks) n×32-entry FIFO



(b) Change in IPC (memory intensive benchmarks) 64 total entries



(c) Change in IPC (other benchmarks) n×32-entry FIFO



(d) Change in IPC (other benchmarks) 64 total entries

Figure 6.6: SCREW performance vs. Bobcat baseline sweeping number of FIFO queues

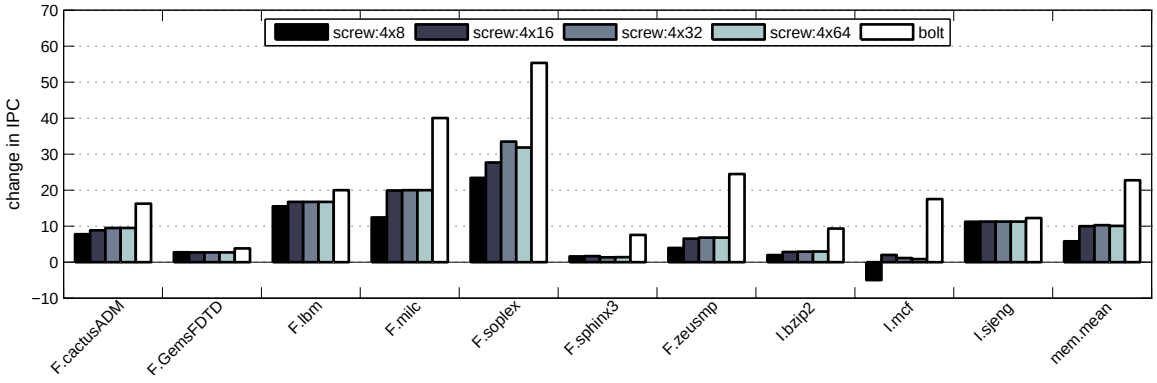
the performance loss experienced by F.soplex, F.sphinx, and I.mcf. The scheduler is able to pick instructions from multiple load chains, ensuring that fewer instructions are 'stuck' behind waiting instructions. We do not see much benefit from increasing the number of FIFO queues beyond four. The average improvement in memory bound workload performance is 9.6% for eight FIFOs compared to 8.3% with four queues. With more queue slots available to dispatching instructions, ROB and register pressure dominate.

Constant Size scaling

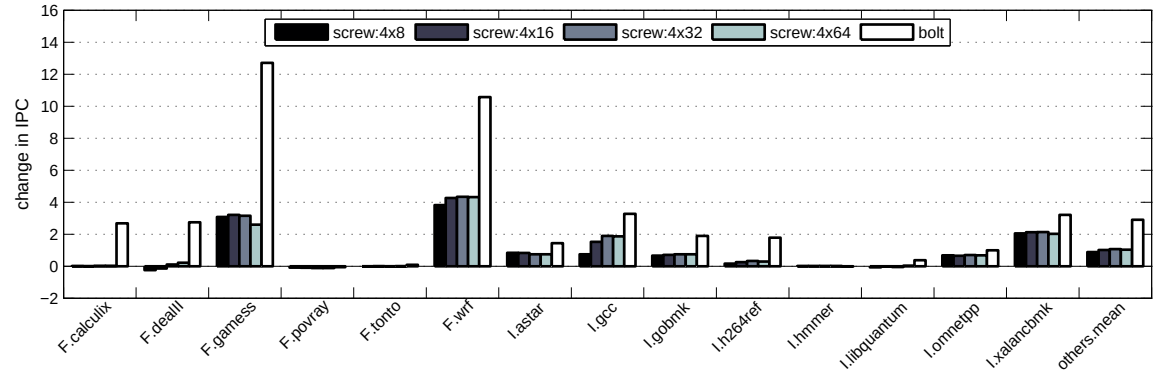
We investigate the effect of FIFO serialization further by keeping the number of FIFO entries constant, while scaling the number of FIFOs. We test three configurations: one 64-entry FIFO, two 32-entry FIFOs, and four 16-entry FIFOs. When an instruction cannot be dispatched into its assigned FIFO, the instruction will stall at dispatch rather than enter another FIFO. Figure 6.6(b) and (d) shows the change in performance while keeping the number of FIFO entries constant. When comparing against Figure 6.6(a) and (c), we see that excess FIFO entries are not needed in the majority of cases. There is little benefit to have four 32-entry FIFOs compared to four 16-entry FIFOs. In both cases, scaling the FIFOs and constant-entry scaling, IPC improves because ILP can be extracted from among independent load chains.

Scaling FIFO Size

Scaling the FIFO size while keeping the number of FIFOs constant will reduce the number of dispatch stalls due to FIFO availability. Figure 6.7 shows the change in performance and average FIFO utilization for a set of four FIFOs, ranging from 8-entries to 64-entries each. We see diminishing returns beyond 16 entries per FIFO because the register file and ROB become limitations. Figure 6.8 shows the average occupancy of all four FIFOs for memory intensive benchmarks. Occupancy is less than 10 entries for all benchmarks except for I.mcf, which has a large number of pointer chasing where loads depend on load-misses. Every configuration reaches the maximum occupancy, causing front end stalls during periods of frequent cache misses. The larger structures suggests that a series of smaller queues should be designed, minimizing area, access-latency, and energy costs.



(a) Change in IPC (memory intensive benchmarks)



(b) Change in IPC (other benchmarks)

Figure 6.7: SCREW performance vs. Bobcat baseline sweeping FIFO size

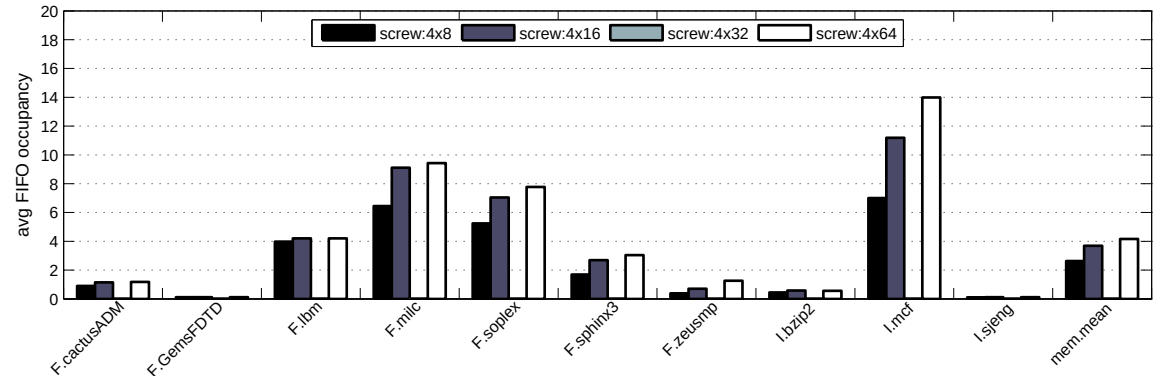


Figure 6.8: Average FIFO utilization (sum of all FIFOs)

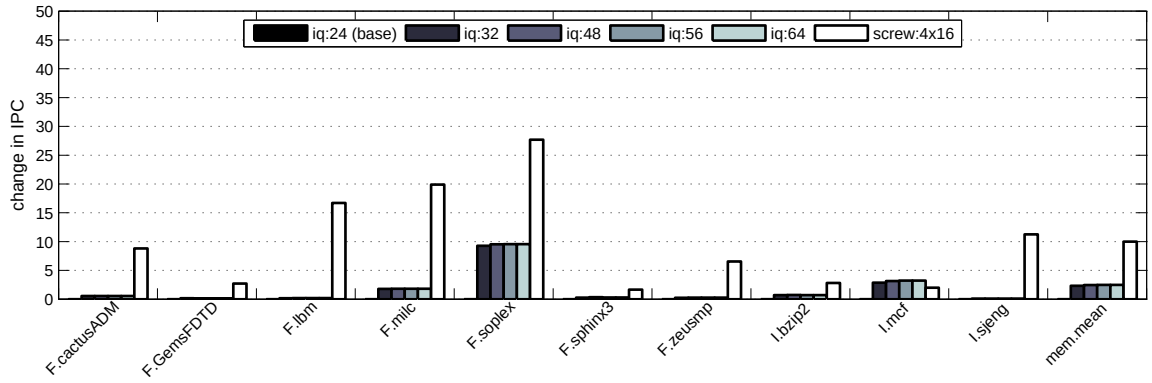


Figure 6.9: Change in IPC scaling baseline issue queue size

6.4.2 Comparing to Base

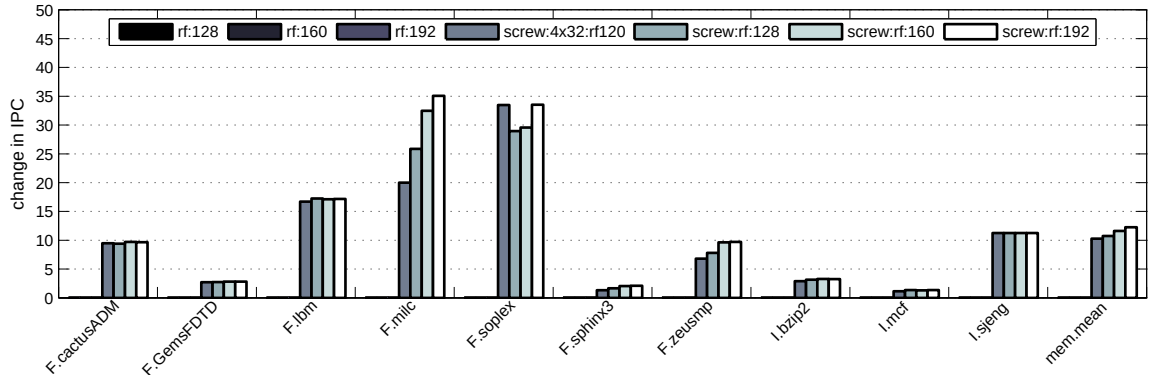
Scaling Baseline Issue Queue

Figure 6.9 shows how the baseline processor behaves when increasing the scheduling queue. We scale the issue queue from the baseline 24-entry queue to a 96-entry queue and compare against a SCREW processor with four 16-entry FIFOs.

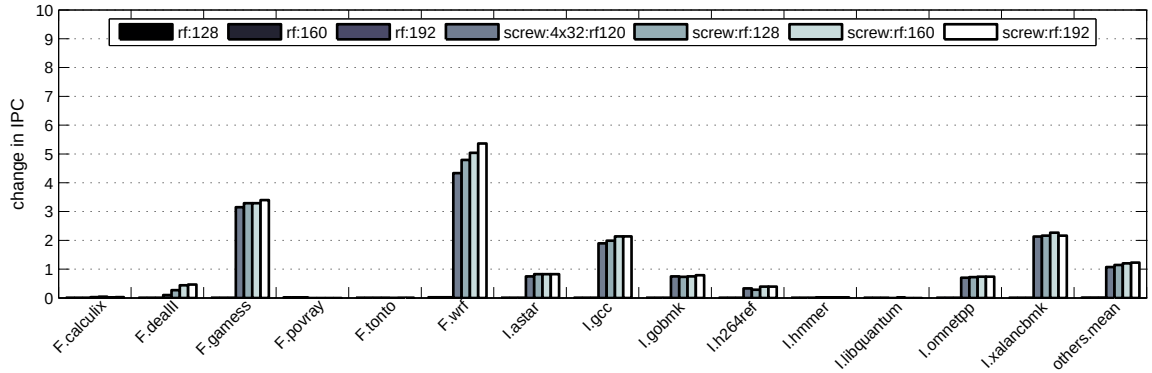
Sensitivity to Register File Size

The performance of both the baseline and SCREW processor can be very sensitive to the number of available registers. Figure 6.10 shows the change in performance when increasing the physical register file size from the base 120-entry register file to a 196-entry register file for both the Bobcat and SCREW processors. Register pressure is particularly important for a SCREW processor because of the instruction lifetime. Instructions which depend on a hit, but are stuck behind instructions dependent on a miss will cause both their register sources *and* their register destination to stay active and alive longer than in an ideally scheduled processor. SCREW will experience increased register pressure because checkpointed instructions do not capture their inputs, allowing source registers to be freed. These sources must stay active throughout the duration of the cache miss. The baseline performance does not change, indicating that the 120-entry register file is sufficiently large for the 56-entry ROB window size.

F.milc is the workload with the greatest sensitivity to register pressure in the SCREW configura-



(a) Change in IPC (memory intensive benchmarks)



(b) Change in IPC (other benchmarks)

Figure 6.10: SCREW processor sweeping RF size

tion. L2 MPKI is sufficiently high that there are significant performance gains from extracting MLP and overlapping load misses. A large instruction window is needed in order to reach the younger load instructions. The intermediate instructions are either executed but not retired (baseline), or executed and speculatively retired (SCREW). The speculatively retired instructions will release over-written registers at execution time, however there are registers 'pinned' to the checkpoint, which reduce the overall capacity of the register file during the speculative region.

6.5 Predictor Bypass

While a sophisticated predictor is needed for accuracy, complexity is problematic for energy efficiency. To reduce the energy impact of the sophisticated predictor, we couple it with a simple, small, bypass table. Without this filter, the load predictor is exposed to all loads that are fetched and

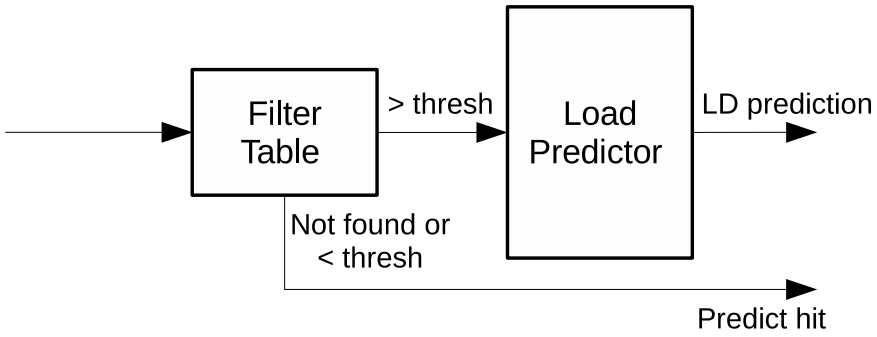


Figure 6.11: Load Predictor Bypass Table

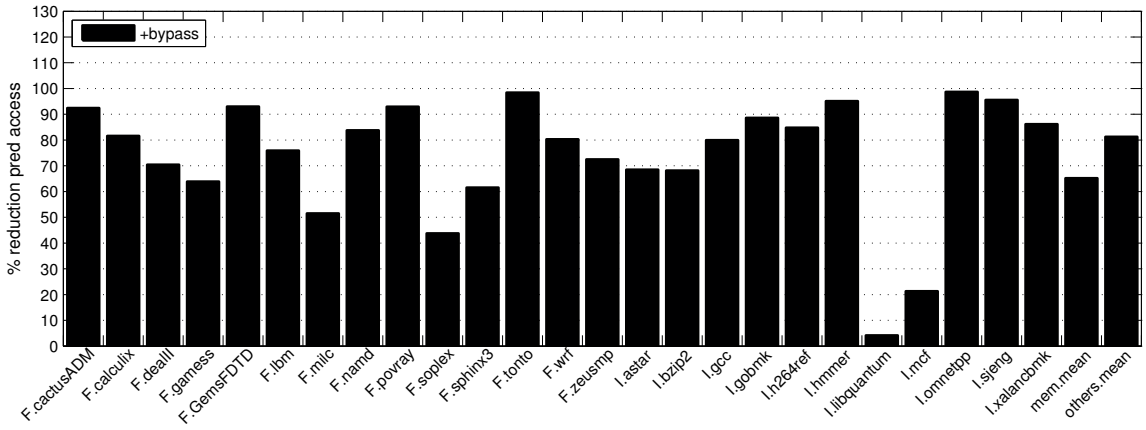


Figure 6.12: % Reduction in Load Predictor Accesses with Bypass Table

dispatched, including those with significant temporal locality that will nearly always hit. Such loads will frequently access and update the predictor with a access-history that is largely dominated by cache hit's, effectively aliasing most instructions to the same few pattern history. It is not worthwhile for a predictor to keep the pattern history for load instructions that will always hit.

The filter is a simple small table, which all loads access to determine if they should access the full predictor. This table is updated with loads that have missed the L2 cache, incrementing a small counter on each miss. Once a threshold has been passed, the load is eligible for prediction. This limits the predictor to a subset of loads that have already missed the cache. The table is built as a small cache, with 64-sets and 2-ways per set for a total of 128 entries. The contents of the predictor-bypass table are replaced as the workload executes, dynamically tracking the set of load instructions that are active at any given point during the workload.

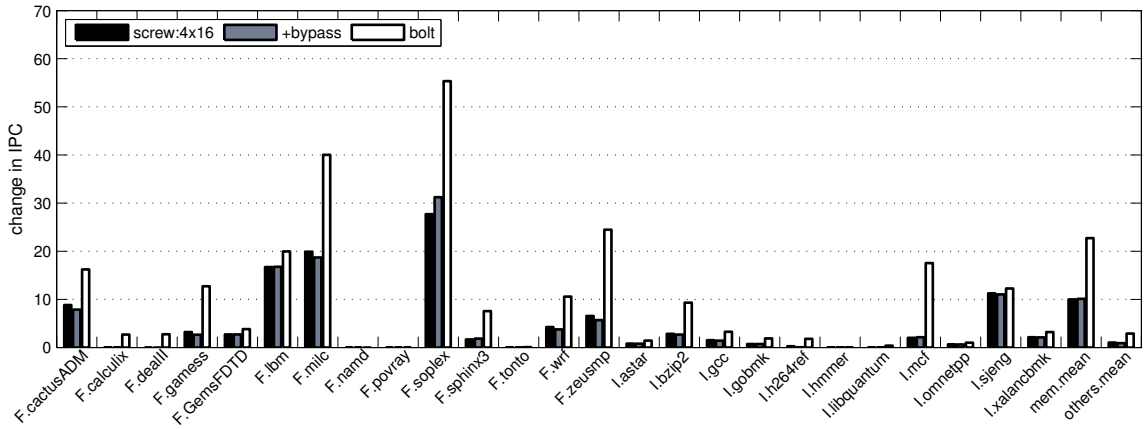


Figure 6.13: IPC with Predictor Bypassing

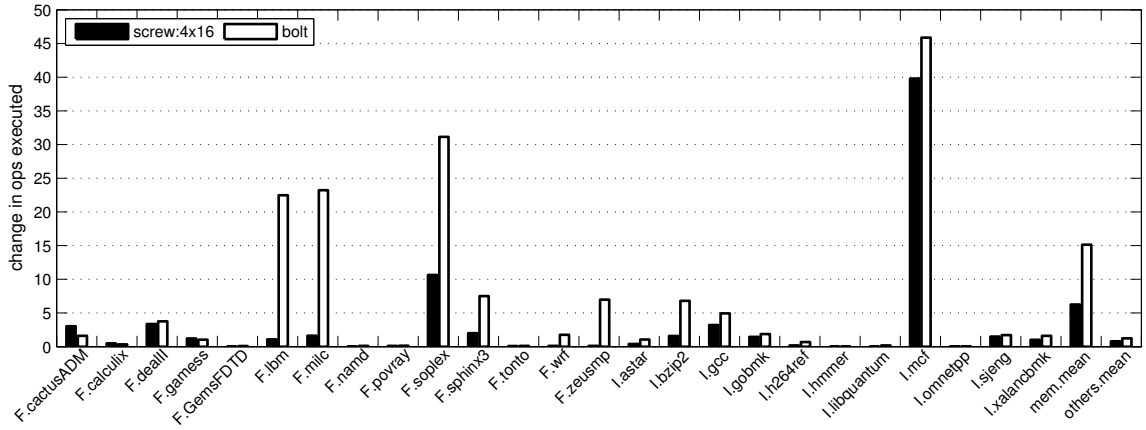


Figure 6.14: Micro-op execution overhead for SCREW and BOLT.

Figure 6.11 illustrates the filter mechanism allowing many loads to bypass the predictor. Load instructions access the table and if an entry matches and is greater than the threshold, then the load is allowed to access the predictor. Load instructions that do not match in the table or match but are not above the filter threshold receive a default 'hit' prediction. The same procedure is followed to update the predictor. When a load misses the L2 cache, it updates the filter table, incrementing the 3-bit counter entry, allocating a new entry if the instruction is not found in the table. Once the counter is at its maximum value, the instruction can update the predictor with its hit or miss behavior. This effectively requires at least eight accesses by the same load instruction before a load hit or miss prediction can be made. Figure 6.12 shows the effect of the filter on both

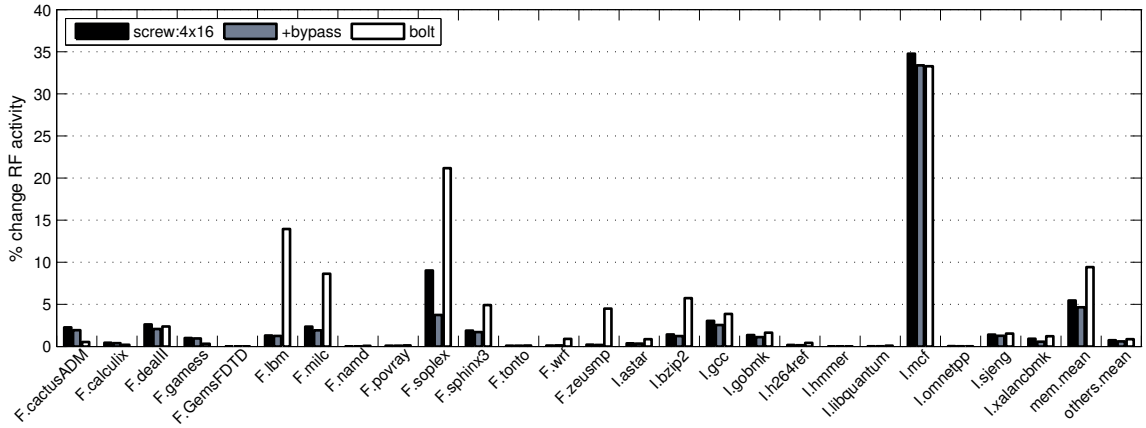


Figure 6.15: RF activity overhead for SCREW and BOLT.

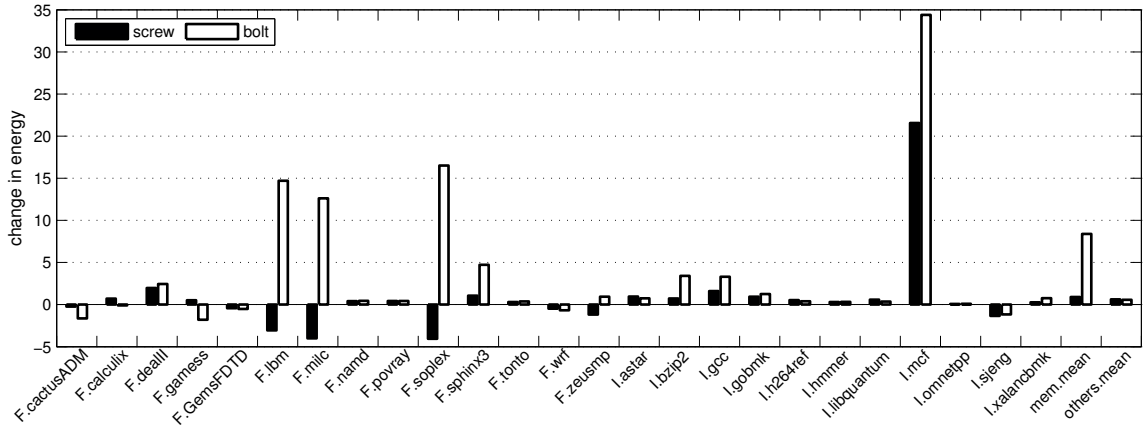


Figure 6.16: Change in Energy

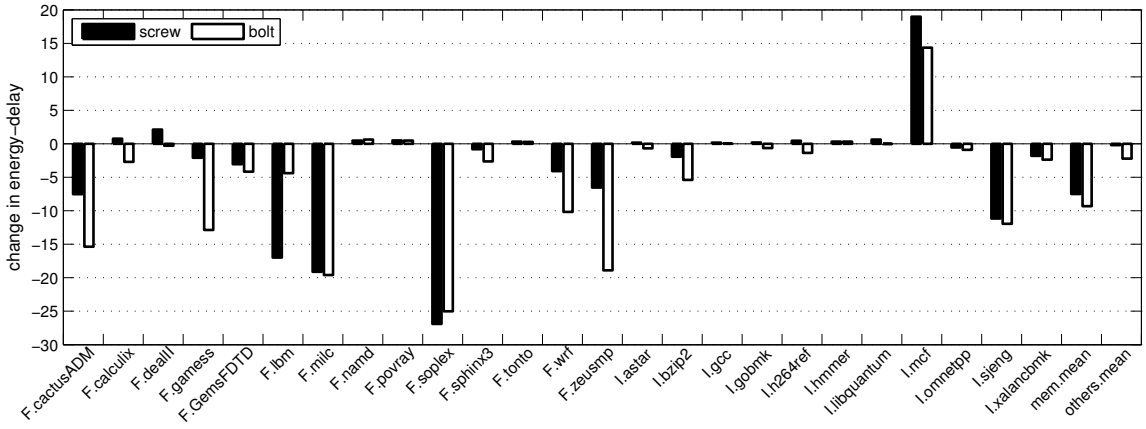


Figure 6.17: Change in Energy-Delay

accuracy compared to an un-filtered predictor. This has little effect on the coverage and accuracy, but reduces the number of accesses to the load-predictor by 65% for memory bound workloads and 80% for compute bound workloads.

Figure 6.13 shows that IPC remains relatively constant across both memory-bound and compute-bound benchmarks. F.cactusADM experiences a 1% reduction in IPC, but has fewer than 10% of the predictor accesses. The average memory bound IPC improvement remains 10%, and the CPU bound improvement remains 2%.

6.6 Energy Costs

Our activity based energy model is described in Section 6.1, model accesses to core structures and the effect of execution time on static energy. A large window processor with coarse grained checkpoint recovery will often execute more instructions for the same workload than a traditional out-of-order processor, as branch mispeculation will cause the pipeline to recover to the start of the checkpoint, rather than to some older instruction still in the pipeline. SCREW has very little execution overhead, with an average of only 6% more instructions than the baseline across memory bound SPEC workloads. SCREW pre-emptively slices instructions to the FIFO, preventing any pipeline loops where a load and its dependents are sliced, executed, and re-sliced due to a subsequent cache miss. This happens often in workloads like I.mcf. Execution overhead increases for memory intensive workloads where the instruction window is large and more checkpoints are created (and therefore more opportunity for the checkpoint to be squashed). BOLT has a 15% increase in micro-op execution over the Bobcat baseline for these memory intensive benchmarks.

Our energy model, described in equation 6.1 and 6.2 characterizes key components of the processor: the RF, issue queue, LT slice components, and remaining energy. Figure 6.17(a) shows the change in energy over the Bobcat baseline for memory bound workloads, with an aggregate column for CPU-bound workloads. SCREW never costs more energy than BOLT across all workloads. I.mcf is an interesting case, where BOLT suffers due to pipeline loops, while SCREW suffers due to predictor energy, as the majority of loads miss in the cache limiting the efficacy of our predictor bypass table. For a few memory bound workloads in SCREW, the improved performance and reduction

in issue queue broadcasts contributes to a net energy reduction. In aggregate, SCREW increases energy by approximately 1% for memory bound workloads, while BOLT increases energy by 8% due to its more expensive slice-out mechanisms.

SCREW and BOLT spend energy in different ways to achieve performance improvements. When slicing an instruction, SCREW does not require an extra register read and value write, which saves RF and slice-buffer energy. SCREW also gates issue-queue broadcasts for FIFO instructions, which reduces power for those instructions with a potential performance cost due to serialization. However, SCREW requires table and predictor accesses which BOLT does not.

Figure 6.17(b) shows the change in energy-delay (ED) for memory intensive workloads. Here, we see BOLT's improved performance brings its energy-delay in line with SCREW. For memory bound workloads, SCREW reduces Energy-Delay by 8%, while Bolt approaches a 10% reduction. Workloads such as `cactus, wrf` and `zeus` where SCREW is not able to correctly predict enough of the load-misses do not achieve the same performance as BOLT. There is negligible change to non-memory bound workloads for both SCREW and BOLT.

Chapter 7: Conclusions

In the multi-core era, attention has shifted from improving the performance of a single execution thread through conventional device and design scaling to focusing on multi-core processors with many execution threads. However, microarchitectures that improve single-threaded performance are still very important and relevant due to constraints in software-level parallelism and thermal dissipation. Amdahl's law still dictates that serial performance will still have a large effect on overall performance, even in embarrassingly parallel workloads. While packaging constraints limit both the number of cores that can be enabled on a processor die and the operating frequency of each core. This dissertation has presented three microarchitectural mechanisms targeting single-threaded performance and energy efficiency, and are still effective in the context of CMP processing environments.

The first technique presented was a detailed implementation of register reference counting. Register reference counting is a replacement for conventional register management, trading a freelist FIFO structure for a bitvector representation. We showed that the area and energy costs are on the same order as the free list, and the mechanism is extensible to support the register management requirements of many performance oriented microarchitectures. Where prior work has relied on register reference counting in an abstract way, we have presented a real implementation and measurement of its energy and area costs, showing that it is a practical replacement for conventional free-list register management.

The second technique presented in this dissertation incorporated our low-cost implementation of register reference counting to intelligently allocate registers to specific portions of the register file, maximizing the opportunity to disable empty register file banks. We presented several different register allocation algorithms along with bank power-gating algorithms. These algorithms target low-ILP regions, where few instructions are executing and the register file resources are not needed. Directed register allocation, coupled with register-file power gating, eliminates the large leakage

overhead from partitioning the register file (done for clock gating), allowing more flexible, banked register file structures to be used with significantly lower leakage overhead.

The final technique presented is a new latency tolerant microarchitecture, SCREW, which applies predictive mechanisms to steer instructions dependent on long-latency cache misses. This microarchitecture depends on the flexible reference counting register management techniques in order to track register usage and effectively scale the register file. This microarchitecture focuses on reducing the energy cost of latency tolerance in several ways: reducing the complexity of the 'slice-out' procedure by eliminating expensive pseudo-execution and value capture, eliminating pipeline loops where instructions are sliced out multiple times by dispatching miss dependent instructions into a FIFO and only releasing when the instructions are ready, and eliminating costly issue-queue wakeup broadcasts by using the same dependency matrix to identify instructions which do not need to broadcast their writeback. The implementation of SCREW in this dissertation scales the instruction window to absorb the latency of cache misses, improving IPC performance up to 30%, and 13% on average for memory intensive workloads with improved energy-delay compared to BOLT.

We show that there is ample room for improvement in the design space of latency-tolerant microarchitectures. We have identified two key inefficiencies of existing approaches. The first is the slice-out procedure, where value-copying is expensive. The second is during the slice-in procedure, where instructions are re-renamed, re-dispatched, and re-executed, but may be sliced out again. The SCREW microarchitecture leverages existing efficiencies from prior works, namely the speculative retirement mechanisms for scaling the ROB and the scalable load and store queues. The keystone of the SCREW microarchitecture is the FIFO-centric design. The in-order FIFO queue and surrounding structures are designed to reduce costly issue queue broadcasts and identify miss-dependent instructions before the cache-miss has occurred.

7.1 Future Work

There are several research threads which can build upon the reference counting and latency-tolerant mechanisms described in this dissertation and address some of the opportunities for improvement:

1. **Examining higher performing predictors.** The predictors used in SCREW looked at only the instruction address of the load. Predictors could also use additional information, such as cache-hints from the ISA or compiler to identify unique or non-unique accesses to memory, or information from the cache prefetcher to identify if data has been brought into the cache. The predictors in this dissertation had trouble with several workloads such as `sphinx` and `zeus`. It is possible that more information from software or from the microarchitecture could improve performance.

2. **Combining latency tolerance with register steering.** The mechanisms presented in this dissertation can dynamically re-size the register file when the excess capacity is not needed or growing it beyond the default size when a cache returns and the miss-dependent instructions drain. The RF can be partitioned for long and short latency instructions, allowing for more banks (*e.g.* a "super-size" mode) to be active for the short period of time after the load returns and the FIFO drains. This dynamic allocation could mitigate the register occupancy problem in SCREW.

3. **Slice out of long latency floating point instructions.** Long latency FP square-root and divide can be identified with 100% accuracy based on their opcode. Dependent instructions can be tracked and sliced out in an identical manner to the predicted load-miss dependent instructions. This should improve performance for workloads with a mixture of integer and floating point instructions.

4. **Critical Path Detection.** The prediction and dependency tracking mechanisms could be combined to act on critical paths, or predicted critical paths [23, 88] If an instruction is predicted or otherwise identified to belong to the critical path of a workload, it could be sliced-out to allow other independent operations to execute around it, while allowing the instruction

window to grow around the critical path if the instructions were treated similar to a long latency operation.

Bibliography

[1] Sandeep R. Agrawal, Valentin Pistol, Jun Pang, John Tran, David Tarjan, and Alvin R. Lebeck. Rhythm: Harnessing data parallel hardware for server workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 19–34, New York, NY, USA, 2014. ACM.

[2] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 423–, Washington, DC, USA, 2003. IEEE Computer Society.

[3] atw.hu. x86,x64 Instruction Latency, Memory Latency, and CPUID dumps. `http://http://instlatx64.atw.hu`.

[4] Steven Battle, Andrew D. Hilton, Mark Hempstead, and Amir Roth. Flexible register management using reference counting. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[5] Steven J. Battle and Mark Hempstead. Register allocation and vdd-gating algorithms for out-of-order architectures. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 108–114, Oct 2013.

[6] Christian Bienia and Kai Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[7] Shekhar Borkar, Tanay Karnik, Siva Narendra, Jim Tschanz, Ali Keshavarzi, and Vivek De. Parameter variations and impact on circuits and microarchitecture. In *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, pages 338–342, New York, NY, USA, 2003. ACM.

[8] Brad Burgess, Brad Cohen, Marvin Denman, Jim Dundas, David Kaplan, and Jeff Rupley. Bobcat: Amd's low-power x86 processor. *IEEE Micro*, 31(2):16–25, March 2011.

[9] Alper Buyuktosunoglu, David Albonesi, Stanley Schuster, David Brooks, Pradip Bose, and Peter Cook. A circuit level implementation of an adaptive issue queue for power-aware microprocessors. In *Proceedings of the 11th Great Lakes Symposium on VLSI*, GLSVLSI '01, pages 73–78, New York, NY, USA, 2001. ACM.

[10] James Charles, Preet Jassi, Narayan S. Ananth, Abbas Sadat, and Alexandra Fedorova. Evaluation of the intel® core™ i7 turbo boost feature. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 188–197, Washington, DC, USA, 2009. IEEE Computer Society.

[11] Zhanping Chen, Mark Johnson, Liqiong Wei, and Kaushik Roy. Estimation of standby leakage power in cmos circuits considering accurate modeling of transistor stacks. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, ISLPED '98, pages 239–244, New York, NY, USA, 1998. ACM.

[12] Niket K. Choudhary, Salil V. Wadhavkar, Tanmay A. Shah, Hiran Mayukh, Jayneel Gandhi, Brandon H. Dwiel, Sandeep Navada, Hashem H. Najaf-abadi, and Eric Rotenberg. Fabscalar: Composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template.

In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 11–22, New York, NY, USA, 2011. ACM.

[13] Adrián Cristal, José F. Martínez, Josep Llosa, and Mateo Valero. A case for resource-conscious out-of-order processors: Towards kilo-instruction in-flight processors. In *Proceedings of the 2003 Workshop on MEmory Performance: DEaling with Applications , Systems and Architecture*, MEDEA '03, pages 3–10, New York, NY, USA, 2003. ACM.

[14] Adrian Cristal, Daniel Ortega, Josep Llosa, and Mateo Valero. Out-of-order commit processors. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, pages 48–, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Adrián Cristal, Oliverio J. Santana, Mateo Valero, and José F. Martínez. Toward kilo-instruction processors. *ACM Trans. Archit. Code Optim.*, 1(4):389–417, December 2004.

[16] John D. Davis, James Laudon, and Kunle Olukotun. Maximizing cmp throughput with mediocre cores. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 51–62, Washington, DC, USA, 2005. IEEE Computer Society.

[17] Eric Donkoh, Teck Siong Ong, Yan Nee Too, and Patrick Chiang. Register file write data gating techniques and break-even analysis model. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, pages 149–154, New York, NY, USA, 2012. ACM.

[18] Ronald G Dreslinski, Michael Wieckowski, David Blaauw, Dennis Sylvester, and Trevor Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, 2010.

[19] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose. Increasing processor performance through early register release. In *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, pages 480–487, Oct 2004.

[20] Oguz Ergin, Deniz Balkan, Dmitry Ponomarev, and Kanad Ghose. Increasing processor performance through early register release. In *in Proceedings of ICCD*, 2004.

[21] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power challenges may end the multicore era. *Commun. ACM*, 56(2):93–102, February 2013.

[22] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[23] Brian Fields, Shai Rubin, and Rastislav Bodík. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 74–85, New York, NY, USA, 2001. ACM.

[24] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 148–157, Washington, DC, USA, 2002. IEEE Computer Society.

[25] J. Friedrich, Hung Le, W. Starke, J. Stuechli, B. Sinharoy, E.J. Fluhr, D. Dreps, V. Zyuban, G. Still, C. Gonzalez, D. Hogenmiller, F. Malgioglio, R. Nett, R. Puri, P. Restle, D. Shan, Z.T. Deniz, D. Wendel, M. Ziegler, and D. Victor. The power8tm processor: Designed for big data, analytics, and cloud environments. In *IC Design Technology (ICICDT), 2014 IEEE International Conference on*, pages 1–4, May 2014.

[26] Dan Gibson and David A. Wood. Forwardflow: A scalable core for power-constrained cmps. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 14–25, New York, NY, USA, 2010. ACM.

[27] Masaharu Goto and Toshinori Sato. Leakage energy reduction in register renaming. In *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04) - Volume 7*, ICDCSW '04, pages 890–895, Washington, DC, USA, 2004. IEEE Computer Society.

[28] Xuan Guan and Yunsi Fei. Reducing power consumption of embedded processors through register file partitioning and compiler support. In *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, pages 269–274, July 2008.

[29] N. Hardavellas, M. Ferdman, B. Falsafi, and A Ailamaki. Toward dark silicon in servers. *Micro, IEEE*, 31(4):6–15, July 2011.

[30] Nikos Hardavellas, Michael Ferdman, Anastasia Ailamaki, and Babek Falsafi. Power scaling: The ultimate obstacle to 1k-core chips. *tech. report NWU-EECS-10-05*, 2010.

[31] Mark Hempstead, Gu-Yeon Wei, and David Brooks. Navigo: An early-stage model to study power-constrained architectures and specialization. In *ISCA Workshop on Modeling, Benchmarking, and Simulations (MoBS)*, Austin, Texas, June 2009.

[32] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[33] Andrew Hilton, Santosh Nagarakatte, and Amir Roth. icfp: Tolerating all-level cache misses in in-order processors. *IEEE Micro*, 30(1):12–19, January 2010.

[34] Andrew Hilton and Amir Roth. Decoupled store completion/silent deterministic replay: Enabling scalable data memory for cpr/cfp processors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 245–254, New York, NY, USA, 2009. ACM.

[35] Andrew Hilton and Amir Roth. Bolt: Energy-efficient out-of-order latency-tolerant execution. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, Jan 2010.

[36] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, et al. The microarchitecture of the pentium® 4 processor. In *Intel Technology Journal*. Citeseer, 2001.

[37] HP. Cacti 6.5. `http://www.hpl.hp.com/research/cacti`.

[38] W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, ISCA '87, pages 18–26, New York, NY, USA, 1987. ACM.

[39] Vijay Janapa Reddi, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. Web search using mobile cores: Quantifying and mitigating the price of efficiency. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 314–325, New York, NY, USA, 2010. ACM.

[40] R. Kalla, B. Sinharoy, and Joel M. Tendler. Ibm power5 chip: a dual-core multithreaded processor. *Micro, IEEE*, 24(2):40–47, Mar 2004.

[41] J. Keller. The 21264: An alpha processor with out-of-order execution. In 9th Annual Microprocessor Forum, Oct. 1996.

[42] Shadi T. Khasawneh and Kanad Ghose. An adaptive technique for reducing leakage and dynamic power in register files and reorder buffers. In Vassilis Paliouras, Johan Vounckx, and Diederik Verkest, editors, *PATMOS*, Lecture Notes in Computer Science, pages 498–507. Springer, 2005.

[43] Yuya Kora, Kyohei Yamaguchi, and Hideki Ando. Mlp-aware dynamic instruction window resizing for adaptively exploiting both ilp and mlp. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 37–48, New York, NY, USA, 2013. ACM.

[44] David Koufaty and Deborah T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56–65, March 2003.

[45] AR. Lebeck, J. Koppanalil, Tong Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 59–70, 2002.

[46] Claude Limousin, Julien Sebot, Alexis Vartanian, and Nathalie Drach-Temam. Improving 3d geometry transformations on a simultaneous multithreaded simd processor. In *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, pages 236–245, New York, NY, USA, 2001. ACM.

[47] Mikko H. Lipasti, Brian R. Mestan, and Erika Gunadi. Physical register inlining. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 325–, Washington, DC, USA, 2004. IEEE Computer Society.

[48] Pierre Michaud. A ppm-like, tag-based branch predictor. In *In Proceedings of the First Workshop on Championship Branch Prediction (in conjunction with MICRO-37)*, 2004.

[49] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 261–270, Washington, DC, USA, 2009. IEEE Computer Society.

[50] T. Monreal, V. Vinals, A Gonzalez, and M. Valero. Hardware schemes for early register release. In *Parallel Processing, 2002. Proceedings. International Conference on*, pages 5–13, 2002.

[51] Teresa Monreal, Antonio González, Mateo Valero, José González, and Victor Viñals. Delaying physical register allocation through virtual-physical registers. In *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 186–192, Washington, DC, USA, 1999. IEEE Computer Society.

[52] Tali Moreshet and R. Iris Bahar. Power-aware issue queue design for speculative instructions. In *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, pages 634–637, New York, NY, USA, 2003. ACM.

[53] Mayan Moudgill, Keshav Pingali, and Stamatis Vassiliadis. Register renaming and dynamic speculation: An alternative approach. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, MICRO 26, pages 202–213, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[54] Todd C. Mowry and Chi-Keung Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 314–320, Washington, DC, USA, 1997. IEEE Computer Society.

[55] M. Mueller, A Wortmann, S. Simon, M. Kugel, and T. Schoenauer. The impact of clock gating schemes on the power dissipation of synthesizable register files. In *Circuits and Systems, 2004. ISCAS '04. Proceedings of the 2004 International Symposium on*, volume 2, pages II–609–12 Vol.2, May 2004.

[56] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 129–, Washington, DC, USA, 2003. IEEE Computer Society.

[57] D.B. Papworth. Tuning the pentium pro microarchitecture. *Micro, IEEE*, 16(2):8–15, Apr 1996.

[58] Dharmesh Parikh, Kevin Skadron, Yan Zhang, Marco Barcella, and Mircea R. Stan. Power issues related to branch prediction. In *In Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, pages 233–244, 2002.

[59] Dharmesh Parikh, Kevin Skadron, Yan Zhang, and Mircea Stan. Power-aware branch prediction: Characterization and design. *IEEE Trans. Comput.*, 53(2):168–186, February 2004.

[60] Il Park, Michael D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, pages 171–182, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[61] Y. N. Patt, W. M. Hwu, and M. Shebanow. Hps, a new microarchitecture: Rationale and introduction. In *Proceedings of the 18th Annual Workshop on Microprogramming*, MICRO 18, pages 103–108, New York, NY, USA, 1985. ACM.

[62] V. Petric, T. Sha, and A. Roth. Reno: a rename-based instruction optimizer. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 98–109, June 2005.

[63] Christian Piguet. *Low-power CMOS circuits - technology, logic design and CAD tools.* CRC Press, 2005.

[64] D.V. Ponomarev, G. Kucuk, O. Ergin, K. Ghose, and P.M. Kogge. Energy-efficient issue queue design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 11(5):789–800, Oct 2003.

[65] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, ISLPED '00, pages 90–95, New York, NY, USA, 2000. ACM.

[66] R.P. Preston, R.W. Badeau, D.W. Bailey, S.L. Bell, L.L. Biro, W.J. Bowhill, D.E. Dever, S. Felix, R. Gammack, V. Germini, M.K. Gowan, P. Gronowski, D.B. Jackson, S. Mehta, S.V. Morton, J.D. Pickholtz, M.H. Reilly, and M.J. Smith. Design of an 8-wide superscalar risc microprocessor with simultaneous multithreading. In *Solid-State Circuits Conference, 2002. Digest of Technical Papers. ISSCC. 2002 IEEE International*, volume 1, pages 334–472 vol.1, Feb 2002.

[67] M. Qazi, M.E. Sinangil, and AP. Chandrakasan. Challenges and directions for low-voltage sram. *Design Test of Computers, IEEE*, 28(1):32–43, Jan 2011.

[68] Steven E. Raasch, Nathan L. Binkert, and Steven K. Reinhardt. A scalable instruction queue design using dependence chains. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 318–329, Washington, DC, USA, 2002. IEEE Computer Society.

[69] Marco A. Ramírez, Adrian Cristal, Alexander V. Veidenbaum, Luis Villa, and Mateo Valero. Direct instruction wakeup for out-of-order processors. In *Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems*, IWIA '04, pages 2–9, Washington, DC, USA, 2004. IEEE Computer Society.

[70] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 199–210, New York, NY, USA, 1997. ACM.

[71] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: Challenges in and avenues for cmp scaling. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 371–382, New York, NY, USA, 2009. ACM.

[72] Amir Roth. Store vulnerability window (svw): Re-execution filtering for enhanced load optimization. In *In International Symposium on Computer Architecture*, pages 458–468, 2005.

[73] Amir Roth. Physical register reference counting. *Computer Architecture Letters*, 7(1):9–12, Jan 2008.

[74] S. Rusu, Simon Tam, H. Muljono, D. Ayers, Jonathan Chang, R. Varada, M. Ratta, and S. Vora. A 45nm 8-core enterprise xeon processor. In *Solid-State Circuits Conference, 2009. A-SSCC 2009. IEEE Asian*, pages 9–12, Nov 2009.

[75] Rama Sangireddy. Register port complexity reduction in wide-issue processors with selective instruction execution. *Microprocess. Microsyst.*, 31(1):51–62, February 2007.

[76] Peter G. Sassone, Jeff Rupley, II, Edward Brekelbaum, Gabriel H. Loh, and Bryan Black. Matrix scheduler reloaded. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 335–346, New York, NY, USA, 2007. ACM.

[77] Toshinori Sato, Hideki Mori, Rikiya Yano, and Takanori Hayashida. Importance of single-core performance in the multicore era. In *Proceedings of the Thirty-fifth Australasian Computer Science Conference - Volume 122*, ACSC '12, pages 107–114, Darlinghurst, Australia, Australia, 2012. Australian Computer Society, Inc.

[78] Tingting Sha, Milo M. K. Martin, and Amir Roth. Scalable store-load forwarding via store queue index prediction. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 159–170, Washington, DC, USA, 2005. IEEE Computer Society.

[79] Tingting Sha, Milo M. K. Martin, and Amir Roth. Nosq: Store-load communication without a store queue. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 285–296, Washington, DC, USA, 2006. IEEE Computer Society.

[80] Youngsoo Shin, Jun Seomun, Kyu-Myung Choi, and Takayasu Sakurai. Power gating: Circuits, design methodologies, and best practice for standard-cell vlsi designs. *ACM Trans. Des. Autom. Electron. Syst.*, 15(4):28:1–28:37, October 2010.

[81] James E. Smith and Andrew R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ISCA '85, pages 36–44, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[82] Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton. Continual flow pipelines. *SIGOPS Oper. Syst. Rev.*, 38(5):107–119, October 2004.

[83] Jared Stark, Mary D. Brown, and Yale N. Patt. On pipelining dynamic instruction scheduling logic. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 57–66, New York, NY, USA, 2000. ACM.

[84] L. Tran, N. Nelson, Fung Ngai, S. Dropsho, and M. Huang. Dynamically reducing pressure on the physical register file through simple register sharing. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '04, pages 78–87, Washington, DC, USA, 2004. IEEE Computer Society.

[85] M. Tremblay and S. Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread cmt sparc processor. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 82–83, Feb 2008.

[86] Jessica H. Tseng and Krste Asanović. Banked multiported register files for high-frequency superscalar microprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 62–71, New York, NY, USA, 2003. ACM.

[87] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, pages 191–202, New York, NY, USA, 1996. ACM.

[88] Eric Tune, Dongning Liang, Dean M. Tullsen, and Brad Calder. Dynamic prediction of critical path instructions. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 185–, Washington, DC, USA, 2001. IEEE Computer Society.

[89] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A modified approach to data cache management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, MICRO 28, pages 93–103, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[90] North Carolina State University. NCSU 45nm Physical Design Kit. `http://www.eda.ncsu.edu/wiki/FreePDK`.

[91] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 430–441, New York, NY, USA, 2011. ACM.

[92] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[93] Kenneth C. Yeager. The mips r10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[94] Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, pages 42–53, Washington, DC, USA, 1999. IEEE Computer Society.

[95] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. A fully integrated multi-cpu, gpu and memory controller 32nm processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 264–266, Feb 2011.

[96] V. Zyuban, J. Friedrich, C. J. Gonzalez, R. Rao, M. D. Brown, M.M. Ziegler, H. Jacobson, S. Islam, S. Chu, P. Kartschoke, G. Fiorenza, M. Boersma, and J.A. Culp. Power optimization methodology for the IBM POWER7 microprocessor. *IBM Journal of Research and Development*, 55(3):7:1–7:9, May 2011.